TESIS DOCTORAL

Paralelización Automática y Estrategias de Desarrollo de Código Eficiente para Aumentar el Rendimiento en Centros de Supercomputación

Javier Corral García

Programa de Doctorado en Tecnologías Informáticas
(TIN)

2021

UNIVERSIDAD DE EXTREMADURA

TESIS DOCTORAL

PARALELIZACIÓN AUTOMÁTICA Y ESTRATEGIAS DE DESARROLLO DE CÓDIGO EFICIENTE PARA AUMENTAR EL RENDIMIENTO EN CENTROS DE SUPERCOMPUTACIÓN

Javier Corral García

con la conformidad de los Directores *

Dr. José Luis González Sánchez y **Dr. Miguel Ángel Pérez Toledano**

* *La conformidad de los directores consta en el original en papel de esta Tesis Doctoral.*

PROGRAMA DE DOCTORADO EN TECNOLOGÍAS INFORMÁTICAS (TIN)

2021

A mi hijo, Javier.

Por ser mi mayor
motivación en esta vida.

Y por el tiempo robado,
que nunca volverá.

Siempre, a mis padres y abuelos.

Por darme todo,
por permitirme llegar hasta aquí.

A Mabel, mi mujer.

Porque sin su comprensión y paciencia
esto no hubiera sido posible.

Por el camino que nos queda por recorrer.

Agradecimientos

A todos los que la presente vieren y entendieren.

Deseo expresar mi sincero agradecimiento a aquellas personas que, de una u otra forma, han colaborado en la consecución de esta tesis doctoral. En especial:

A los directores de toda la investigación, los profesores José Luis González Sánchez y Miguel Ángel Pérez Toledano, por su apoyo y ayuda durante estos años.

A la Fundación COMPUTAEX y el equipo de ingenieros e investigadores de CénitS, el centro de supercomputación donde desarrollo mi carrera profesional desde hace más de una década y en cuyos recursos computacionales se refrendan los resultados alcanzados. Especialmente a Felipe Lemus Prieto, por su inestimable implicación en todos los experimentos relacionados con el consumo de energía. También a Jesús Calle Cancho, por su asistencia y profesionalidad como administrador de sistemas del centro.

A la Doctora Marisol Sánchez Alonso, que me apoyó en mis primeros pasos hacia este objetivo y a quien siempre recordaré con cariño, en particular por su inefable compromiso y dedicación. Allá donde estés... ¡Lo conseguimos!

A mi familia, en especial a mis padres y mi mujer, por comprender y apoyar mi objetivo en todo momento. Partícipes de mi travesía como doctorando, conocen mejor que nadie el empeño implícito entre las páginas de este libro.

Y por supuesto, a mi hijo, que con su corta edad ha sabido esperarme pacientemente en incontables ocasiones.

Resumen

Expertos de múltiples ramas del conocimiento se enfrentan diariamente a multitud de desafíos en proyectos científicos, técnicos o industriales que requieren el uso de la computación de alto rendimiento (HPC, *High-Performance Computing*) para satisfacer adecuadamente sus necesidades. Sin embargo, el desarrollo de algoritmos y programas que empleen correctamente este tipo de infraestructuras implica un importante conocimiento previo que muchos de estos usuarios no poseen, lo cual dificulta notablemente su labor.

Este tipo de proyectos, ampliamente multidisciplinares y heterogéneos, demandan una obtención de resultados con tiempos de ejecución realmente críticos, necesitando para ello la imprescindible utilización de la computación paralela. De este modo, con objeto de alcanzar resultados confiables y emplear eficientemente los recursos de cómputo disponibles, muchos de estos profesionales necesitan una importante ayuda adicional que les permita llevar a cabo la paralelización y optimización de sus códigos. De hecho, es habitual que el primer obstáculo al que se enfrenten sea precisamente la adaptación de sus propios algoritmos secuenciales. En algunos casos, implementar estos códigos ya resulta una tarea complicada para expertos de diversas ramas de la ciencia muy alejadas de la programación informática, lo cual prolonga significativamente su curva de aprendizaje. La situación empeora especialmente cuando precisan además el uso de la computación paralela. De esta forma, para estos usuarios obtener una solución adecuada puede convertirse en un objetivo de notable complejidad.

Este problema está principalmente motivado por el hecho de que los algoritmos paralelos no son diseñados únicamente para realizar el trabajo computacional, sino que también indican cómo será distribuido o dividido éste entre múltiples unidades de procesamiento. Así, es común que científicos e investigadores, que carecen de la formación específica requerida, empleen demasiado tiempo en desarrollar y ejecutar correctamente este tipo de algoritmos. Esto propicia además que sus códigos no sean tan eficientes como deberían, para lo cual se requiere mayor preparación y experiencia.

La completa paralelización automática de códigos de programación, considerando además su eficiencia y el correcto uso de los recursos de cómputo disponibles, sería de utilidad, no solo para estos investigadores, sino también para los propios administradores de los centros de computación de alto rendimiento. De esta forma, los usuarios podrían hacer un uso más eficiente tanto de su propio tiempo, como de las infraestructuras ofrecidas por estos centros, donde dicha eficiencia es clave en diversos aspectos, entre los que destacan: los tiempos de cómputo y de espera para acceder a las distintas infraestructuras,

el número de núcleos o procesadores empleados y la disponibilidad de memoria y espacio en disco, sin olvidar el consumo energético, que actualmente supone un desafío clave para los centros HPC.

En consecuencia, uno de los objetivos principales de la presente tesis doctoral consiste en dar solución a los problemas identificados, proponiendo un transcompilador (compilador *source-to-source*) para la paralelización automática de códigos secuenciales, con el cual será posible obtener mejores rendimientos y eficiencias en las ejecuciones y escoger una correcta estrategia de planificación para cada código. De esta manera, el transcompilador determina las partes del código que pueden ser paralelizadas y genera automáticamente la correspondiente versión paralela. La realización de transformaciones *source-to-source* secuencial-paralelo permite además a los usuarios comparar ambos códigos de forma más simple, facilitando el aprendizaje y la mejora de sus capacidades en programación paralela.

Adicionalmente, es habitual que los programadores centren sus esfuerzos en aquellas instrucciones que pueden ser paralelizadas, sin tener en cuenta la eficiencia del resto del código, a menudo ignorando el importante efecto que las partes secuenciales tienen sobre los tiempos de ejecución. El impacto asociado es especialmente significativo en trabajos HPC que tardan varios días en ejecutarse o que forman parte de proyectos que requieren miles de horas de CPU anuales, algo muy común en variedad de investigaciones científicas. Por ello, esta tesis evalúa y analiza diversas técnicas *software* con el objetivo de conseguir mejoras adicionales en el código y reducir los tiempos de ejecución. Las técnicas han sido seleccionadas entre la literatura existente y escogidas por ser las más representativas o que mejoran de modo más notable la eficiencia, entendiendo ésta como el rendimiento computacional alcanzado por un programa en relación al alcanzable en una situación óptima. En primer lugar, se realiza un detallado análisis del impacto producido al aplicar las distintas estrategias en dispositivos IoT (*Internet of Things*, Internet de las cosas), debido a su mayor sencillez para realizar todas las mediciones necesarias. Tras comprobar su impacto en la reducción de los tiempos de ejecución y en el ahorro energético, las mismas estrategias son aplicadas sobre infraestructuras HPC. De este modo se demuestra que las técnicas propuestas permiten aumentar la eficiencia de forma fácil y sencilla en ambos entornos, posibilitando que los usuarios obtengan destacables mejoras de rendimiento con cambios menores en sus códigos.

Con todo lo anterior se consigue que incluso usuarios noveles puedan hacer un uso más apropiado y eficiente de los recursos computacionales, disminuyendo los problemas inherentes al aprendizaje de la programación paralela.

Índice

Índice de figuras

Índice de Tablas

Índice de códigos

Capítulo 1

Introducción

En este primer capítulo se detallan los antecedentes de la investigación desarrollada y se identifican las motivaciones previas existentes, definiendo los objetivos planteados y resumiendo las distintas propuestas y aportaciones de la tesis. Asimismo, en el último apartado se describen las secciones que desglosan el presente documento.

1.1. Antecedentes

En los últimos años ha crecido exponencialmente el número de retos que requieren el uso de la computación de alto rendimiento (HPC, *High-Performance Computing*), tanto en el ámbito académico como en la industria [1–6]. Para acometerlos y dar respuesta a los desafíos críticos planteados, científicos e investigadores necesitan habitualmente conocimientos de dominio sumamente específicos relacionados con la implementación y ejecución de algoritmos y códigos paralelos. Sin embargo, la curva de aprendizaje puede resultar especialmente exigente para expertos con especializaciones muy alejadas de las tecnologías de la información y la comunicación, así como para profesionales de distintas áreas que únicamente han implementado programas secuenciales y carecen de las destrezas necesarias para interactuar con los recursos disponibles. Para este tipo de usuarios, obtener una solución adecuada mediante HPC puede convertirse en un objetivo complicado de alcanzar [7].

Además, es frecuente que ciertos usuarios de estas infraestructuras se preocupen únicamente por adaptar sus códigos para que estos puedan ejecutarse de forma paralela, sin importarles ni tener en consideración la eficiencia de estos, principalmente por no ser conscientes del impacto provocado sobre sus ejecuciones e ignorar las posibilidades y limitaciones reales de los recursos utilizados. De hecho, no suelen necesitar que sus algoritmos sean especialmente eficientes, debido a las características ofrecidas por este tipo de infraestructuras, cuya potencia y gran rendimiento computacional permiten obtener resultados en tiempos realmente significativos. Sin embargo, esto acaba suponiendo un problema para los centros de HPC y sus administradores de sistemas, donde el uso adecuado de los recursos y un consumo energético eficiente son siempre desafíos clave

[8]. Así, cada centro aplica sus propias estrategias para afrontar estos problemas y ayudar a sus usuarios. Con este fin pueden, por ejemplo, proporcionar herramientas e interfaces que faciliten la utilización de la computación paralela [9–14] u ofrecer cursos de formación específicos centrados en las capacidades y los conocimientos necesarios.

La presente tesis doctoral ha sido desarrollada en CénitS [15], el Centro Extremeño de iNvestigación, Innovación Tecnológica y Supercomputación, que promueve y presta servicios de HPC a la comunidad investigadora, proporcionando las infraestructuras, los recursos y el soporte técnico necesarios para llevar a cabo proyectos científicos, técnicos y empresariales que requieran el uso del cálculo intensivo y las comunicaciones avanzadas o la aplicación de paradigmas tecnológicos como *Big Data*, *Cloud Computing* o *Machine Learning*, entre otros. Los ingenieros que trabajan en este tipo de centros ayudan a sus usuarios a ejecutar proyectos de I+D+i que persiguen aportar soluciones innovadoras en desafíos sociales, medioambientales y científico-tecnológicos, facilitando así el progreso científico. De este modo, ofrecen soporte a líneas de investigación realmente heterogéneas que aportan soluciones en ámbitos tan diversos como: agricultura y ganadería de precisión; eficiencia energética; impacto medioambiental; predicción climática; secuenciación genética y biomedicina; virtualización de puestos de trabajo; química computacional; monitorización y telemetría para infraestructuras inteligentes; predictibilidad de flujos de tráfico en ciudades inteligentes; o simulaciones electromagnéticas, entre otros [16–18].

Numerosos usuarios, a menudo científicos e investigadores, que utilizan los recursos de los centros HPC no son programadores experimentados. Sin embargo, es común que necesiten adaptar códigos secuenciales ya existentes al entorno computacional concreto sobre el que utilizan sus programas, lo cual requiere una clara comprensión del correspondiente paralelismo. Pese a la ausencia de estos conocimientos, los usuarios desean ejecutar sus códigos utilizando el mayor número posible de núcleos o unidades de procesamiento, creyendo erróneamente que esto siempre les permitirá obtener mejores resultados. Adicionalmente, dado que la ejecución de trabajos en estas infraestructuras de cómputo implica deber estimar de antemano los valores máximos de horas de CPU a emplear, número de procesadores o núcleos y espacio en disco requeridos por cada proyecto, esto provoca que no se haga un uso eficiente de los recursos, afectando negativamente a los tiempos medios de espera en las colas de ejecución a las que se envían estos trabajos.

Actualmente, la paralelización automática de códigos secuenciales es considerada un desafío clave por la comunidad HPC [19], siendo objeto de diversas líneas y proyectos de investigación [19–30], debido principalmente a la complejidad que supone su aplicación a gran escala [8]. De este modo, aunque en la literatura se recogen diferentes propuestas para la paralelización de códigos sin requerimientos de interacción por parte del usuario [20, 24, 25], no todas las necesidades identificadas en los centros de computación de alto rendimiento se ven cubiertas [31]. En muchos casos, el principal problema reside en el hecho de que, al enfrentarse a la programación paralela, los usuarios suelen presentar considerables dificultades para comprender cómo mejorar la eficiencia de sus códigos, más allá de la propia paralelización de los mismos o la utilización de las optimizaciones automáticas ofrecidas por los compiladores [13, 14, 32].

1.2. Motivaciones y objetivos

La correcta utilización de la computación de alto rendimiento precisa de notable experiencia y aprendizaje previos, para adquirir las múltiples y complejas capacidades asociadas a su uso [33]. El desarrollo de códigos que aprovechen eficientemente el rendimiento de los sistemas HPC requiere a su vez conocimientos específicos sobre algoritmos paralelos, especialmente en términos de precisión, eficiencia y velocidad de ejecución. Por ello, su adecuada utilización suele resultar compleja, especialmente para usuarios noveles en programación que necesitan paralelizar sus códigos secuenciales. Así, es habitual que estos usuarios, que carecen de la formación específica requerida, empleen demasiado tiempo en desarrollar correctamente algoritmos que aprovechen realmente las ventajas que ofrece el uso de la computación paralela. Además, es común que ejecuten sus desarrollos sin preocuparse por la eficiencia de sus códigos ni tener en cuenta que otros usuarios permanecen en espera para acceder a los recursos que ellos están utilizando de forma inapropiada.

Por ello, el primer objetivo de esta investigación consiste en ofrecer un marco de trabajo que permita hacer un uso más eficiente de los recursos ofrecidos por los centros de computación de alto rendimiento, reduciendo los tiempos de ejecución de sus códigos y ayudando a facilitar el aprendizaje inicial de la programación paralela. De esta forma, los usuarios podrían dedicar más esfuerzo a su tarea principal que, en la mayoría de los casos, es investigar en sus respectivos campos, no desarrollar código. Sin embargo, los distintos enfoques dedicados a la paralelización automática de códigos secuenciales propuestos en la literatura [19–30], no resuelven aún todas las necesidades identificadas en los centros de HPC [31], donde muchos usuarios buscan ejecutar sus códigos secuenciales y paralelos sin considerar previamente su optimización.

Por otro lado, los compiladores proporcionan técnicas de bajo nivel que permiten disminuir el tamaño del código, utilizar menos memoria, reducir el tiempo de ejecución o realizar un número inferior de operaciones de entrada/salida al transformar el código fuente en instrucciones en lenguaje máquina. Su objetivo es por tanto, alcanzar estas ventajas conservando la semántica del programa, de modo que el algoritmo original se mantenga sin cambios. Estas ayudas ofrecidas por los compiladores suponen un recurso ampliamente más utilizado que la aplicación de técnicas manuales por parte de los propios programadores. Sin embargo, como manifiesta esta tesis, las habilidades de programación desempeñan un importante papel en este sentido, dado que el programador ejerce un gran impacto en la eficiencia del código, demostrándose que en algunos escenarios los compiladores no obtienen las mismas mejoras que puede lograr un programador mediante la aplicación manual de técnicas de optimización.

A lo anterior hay que añadir el hecho de que acelerar ciertas ejecuciones también puede conducir a la consecución de mayores eficiencias energéticas. Así, aunque el consumo de energía también depende de una variable de potencia, disminuir los tiempos de ejecución implica reducciones del consumo cuando dicha potencia es constante en el tiempo. Asumiendo esto, varios trabajos han demostrado que el uso de la energía y el tiempo están directamente relacionados [34], especialmente respecto a lenguajes como C y C ++ [35, 36], y concluyen que en relación a la eficiencia energética es preferible

implementar códigos que se ejecuten en el menor tiempo posible [37]. Aunque esto no puede considerarse una regla general, ya que las disminuciones en los tiempos de ejecución no siempre implican reducciones de consumo.

Por ello, el segundo objetivo de esta investigación está centrado en la escritura de códigos más eficientes[1] y persigue que los usuarios puedan aumentar de forma significativa el rendimiento de sus programas, reduciendo los tiempos de ejecución y el consumo energético, gracias a la aplicación de un conjunto de técnicas *software*. De este modo, se contribuye además a que los programadores sean conscientes del notable impacto que la aplicación de ciertas técnicas, en pequeños y simples fragmentos de sus códigos, puede tener sobre la eficiencia. Las mejoras en el rendimiento son especialmente significativas en lo relativo a trabajos de HPC que tardan días o incluso semanas en completarse o que forman parte de proyectos que requieren miles de horas de CPU al año, algo realmente común en una amplia variedad de investigaciones científicas. En este sentido, es muy importante que sean conscientes del excesivo tiempo que puede malgastar un pequeño fragmento de código ineficiente cuando su ejecución es repetida durante innumerables ocasiones. Adicionalmente, es habitual que cuando programadores no expertos se enfrentan a la programación paralela presenten notables dificultades para optimizar sus códigos, aunque puedan identificar correctamente las partes donde la ejecución requiere más tiempo de proceso gracias a *profilers* o herramientas de análisis [38–40]. Además, estos programadores suelen centrar sus esfuerzos únicamente en las instrucciones que se pueden paralelizar, sin tener en cuenta la eficiencia de las partes secuenciales y, a menudo, ignorando el considerable impacto que éstas tienen en el tiempo final de ejecución o en el consumo energético asociado.

Sin embargo, en la literatura no existen demasiadas propuestas destinadas a optimizar los tiempos de ejecución (secuenciales y paralelos) mediante la aplicación de técnicas *software* que posibiliten la escritura de código eficiente. El presente trabajo demuestra además, que dicho problema afecta tanto a grandes infraestructuras de HPC, como a pequeños dispositivos como los empleados en el Internet de las cosas (IoT, *Internet of Things*).

1.3. Propuestas y contribuciones

Esta tesis persigue cubrir las necesidades identificadas respecto a la supercomputación proponiendo, en primer lugar, un transcompilador (compilador *source to source*, fuente a fuente) para mejorar el rendimiento en centros HPC, de forma que incluso usuarios menos avanzados puedan utilizar adecuadamente sus recursos, mejorar sus ejecuciones y ahorrar tiempo en el propio desarrollo de sus programas paralelos. De este modo, está principalmente enfocado a ayudar a quienes carezcan de la experiencia previa necesaria en la utilización de la programación paralela, posibilitando la paralelización automática de sus códigos secuenciales. Además, ofrece una interfaz gráfica que facilita su uso, y permite a sus usuarios identificar las partes secuenciales y paralelas de forma sencilla, favoreciendo

[1] Entendiendo como código eficiente aquel que obtiene un rendimiento similar al alcanzable en una situación óptima.

el aprendizaje y la mejora de sus habilidades de programación y ayudando a disminuir la curva de aprendizaje correspondiente al uso de este tipo de algoritmos.

El transcompilador permite la paralelización automática de programas secuenciales escritos en lenguaje C/C+, mediante análisis estático (con características conocidas en tiempo de compilación), ofreciendo como salida un programa OpenMP (*Open Multi-Processing*) [9] equivalente. En concreto, focaliza su objetivo en los bucles, en los cuales los programas que explotan el paralelismo emplean la mayor parte de su tiempo, especialmente en proyectos relacionados con la ciencia o la ingeniería [41].

También se propone un sistema de apoyo a la toma de decisiones (DSS, *Decision Support System*), que posibilita la creación de conjuntos de datos para el aprendizaje supervisado empleado por el transcompilador y favorece el análisis de los diferentes factores que influyen en el rendimiento y la eficiencia. Este DSS permite modificar, compilar, ejecutar y evaluar automáticamente los resultados obtenidos por las ejecuciones de los programas paralelos. Adicionalmente, presenta otras características como el envío automático de ejecuciones a las infraestructuras HPC de ejecución o la extracción, la ordenación y el análisis de la información obtenida para cada código. Esto facilita significativamente seleccionar aquellos programas que presentan unos parámetros adecuados de paralelización en función de una carga de trabajo o una eficiencia determinadas.

Además, el transcompilador ha sido desarrollado con el propósito de que en el futuro pueda ser ampliado con un nuevo módulo, que permita realizar optimizaciones adicionales de fragmentos secuenciales y paralelos del código de salida, con objeto de mejorar las reducciones en los tiempos de ejecución. Para lograr este objetivo, se han evaluado diferentes técnicas de optimización para escribir programas más eficientes. De este modo, se demuestra que su integración en el transcompilador ayudaría a mejorar el rendimiento del código.

Así, otra contribución de esta investigación consiste en proponer una serie de técnicas *software* para la escritura eficiente de código, que posibilitan reducir tanto el tiempo de ejecución como su consumo energético asociado, sin variar la semántica de los algoritmos originales ni alterar sus resultados. Estas técnicas permiten a los programadores obtener significativas mejoras realizando pequeñas modificaciones en sus programas. Asimismo, este trabajo ofrece un extenso análisis sobre su uso.

Los tiempos obtenidos mediante la aplicación manual de estas técnicas son comparados con los alcanzados mediante las optimizaciones automáticas ofrecidas por ciertos compiladores. Los experimentos realizados demuestran que estas últimas no son suficientes, de modo que la influencia del programador en la eficiencia del código es decisiva y gracias a la utilización de las técnicas propuestas es posible incrementar notablemente las mejoras obtenidas por estos compiladores. El impacto de las técnicas es analizado tanto en infraestructuras HPC como en dispositivos del Internet de las cosas. De este modo, se identifican importantes aspectos relativos a la optimización de código que serán clave en las futuras ampliaciones del transcompilador, de forma que éste, además de realizar paralelizaciones automáticas, también ayude al usuario a programar de manera eficiente.

Asimismo, otra de las contribuciones se centra en analizar el ahorro que puede obtenerse al utilizar las técnicas de optimización recogidas en esta tesis, demostrando su eficacia para escribir código energéticamente eficiente[2] y constatando que, también en este caso, la ayuda automática que ofrece el compilador es insuficiente en términos de ahorro energético. De este modo, se evalúa la energía consumida al aplicar las técnicas y se realizan comparaciones respecto a los resultados obtenidos mediante el uso del compilador. Así, se demuestra que los programadores pueden ahorrar más energía aplicando manualmente las técnicas propuestas.

1.4. Estructura de la tesis

A continuación se describen las sucesivas secciones que conforman el presente trabajo:

El Capítulo 2 proporciona una contextualización de la computación de alto rendimiento y la programación paralela, describiendo conceptos y herramientas fundamentales de ambas. Asimismo, ofrece una visión general sobre las distintas perspectivas aplicadas para afrontar los desafíos que presenta este modelo de programación. Además, resume las propuestas recogidas en la literatura sobre paralelización y aprendizaje automáticos, así como aproximaciones relacionadas con la escritura de códigos de programación eficientes.

Los Capítulos 3 y 4 se centran en las principales aportaciones de la tesis: el tercero presenta el transcompilador desarrollado para la paralelización automática de códigos de programación, mientras que en el cuarto se describen las técnicas *software* seleccionadas y analizadas para el desarrollo de códigos eficientes.

En el Capítulo 5 se presentan, en primer lugar, los resultados experimentales obtenidos en relación al transcompilador, analizando su eficacia para paralelizar automáticamente programas secuenciales. A continuación se debate la aplicación de las técnicas propuestas, evaluando sus posibilidades para mejorar la eficiencia de los códigos de programación, respecto a la reducción de los tiempos de ejecución y al ahorro del consumo energético.

El Capítulo 6 recoge las conclusiones alcanzadas, las limitaciones actuales y el trabajo futuro a desarrollar.

Además, se ofrecen los siguientes anexos: el Apéndice A resume las publicaciones del autor; en el Apéndice B pueden verse los bucles paralelizados de forma automática en los experimentos desarrollados 5; y el Apéndice C contiene los tests diseñados e implementados *ad hoc* para la evaluación de las técnicas eficientes propuestas.

Al final del libro pueden encontrarse las referencias bibliográficas y el listado de acrónimos.

[2]El concepto de eficiencia energética puede entenderse como la energía consumida en relación con la mínima necesaria en una situación óptima. Aunque inicialmente fue centrado mayoritariamente en el *hardware*, en la actualidad supone una preocupación realmente importante para desarrolladores de todo tipo de *software* [10]. De hecho, se ha convertido en un intenso campo de investigación que presenta como principal objetivo analizar y optimizar el consumo energético de estos sistemas.

Capítulo 2

High-Performance Computing y programación paralela

Este capítulo ofrece, en primer lugar, una detallada introducción a la computación de alto rendimiento y la programación paralela, resumiendo sus conceptos más importantes así como sus herramientas más significativas. De igual modo, también se enmarcan las diversas perspectivas que permiten afrontar los desafíos inherentes a este modelo de programación. Además, se describen las principales propuestas existentes en la literatura enmarcadas en la paralelización y el aprendizaje automáticos, exponiendo asimismo las aproximaciones publicadas en relación a la escritura de códigos de programación eficientes.

2.1. High-Performance Computing

La computación de alto rendimiento, también conocida como supercomputación, puede ser definida como aquella que alcanza un rendimiento ampliamente superior al ofrecido por la mayoría de ordenadores de escritorio o portátiles actuales, con el fin de resolver grandes problemas científicos o de ámbito ingenieril o empresarial [42]. Gracias a la supercomputación es posible obtener respuestas a preguntas enmarcadas en una extensa variedad de desafíos que no pueden ser resueltos de forma empírica, teórica o a través de la inmensa mayoría de los ordenadores comerciales [33] debido, entre otros factores, a requisitos de almacenamiento, memoria o cómputo. Es uno de los desarrollos más importantes de la era moderna, presentando un impacto inigualable en gran variedad de campos de investigación y desempeñando un papel esencial en el progreso de la humanidad [33]. De hecho, actualmente resulta clave en ámbitos como la medicina, la anticipación de fenómenos meteorológicos extremos, la monitorización del cambio climático, la ciberseguidad y la protección de infraestructuras críticas, el desarrollo y la innovación industrial y la transformación digital, entre otros [43].

La Comisión Europea ha establecido una estrategia específica para apoyar el HPC, al considerarlo una tecnología habilitadora para la ciencia, la industria y la sociedad, con el principal objetivo de convertir a Europa en una potencia mundial en supercomputación [44]. De este modo, en 2018 se creó la empresa común EuroHPC JU (*European HPC joint undertaking*) [45], una iniciativa conjunta entre la Unión Europea, países europeos y socios privados, para desarrollar un ecosistema de supercomputación de clase mundial en el continente. De este modo, la EuroHPC JUpermite a las naciones participantes coordinar sus esfuerzos y poner en común sus recursos con el objetivo de desplegar en Europa supercomputadores a exaescala de clase mundial, capaces de realizar más de un billón (10^{18}) de operaciones por segundo. También pretende impulsar el desarrollo de tecnologías y aplicaciones innovadoras en supercomputación, mejorando la cooperación en investigación científica avanzada, impulsando la competitividad industrial y asegurando la autonomía tecnológica y digital europea [45].

Las operaciones en coma flotante por segundo, FLOPS, suponen la métrica de rendimiento más utilizada en HPC, aunque en realidad no hay una única medida que refleje adecuadamente todos los aspectos de un supercomputador [33]. No obstante, los sistemas de supercomputación existentes son clasificados regularmente mediante múltiples rankings basados en resultados de *benchmarks*, como HPCG [46], HPC-AI [47, 48] y Graph 500 [49]. Uno de los índices de referencia más populares es el Top 500 [50], que ofrece semestralmente una lista actualizada con la clasificación de los sistemas de supercomputación más potentes del mundo según su rendimiento en el *benchmark* Linpack [51, 52]. Éste es independiente de la arquitectura del supercomputador y está basado en la resolución de ecuaciones lineales de tipo "$Ax = b$" mediante matrices aleatorias densas y aritmética de punto flotante. Este ranking muestra información sobre cada sistema, incluyendo datos sobre el fabricante, el número de procesadores y núcleos, la memoria, el consumo energético o el sistema operativo, entre otras características. El *benchmark* utilizado se centra únicamente en el rendimiento máximo de ejecución en operaciones de coma flotante para un sistema de ecuaciones concreto, sin estresar la memoria o la red interna. Sin embargo, su simplicidad y facilidad de uso han originado el dominio de este listado frente a la utilización de otras alternativas mucho más completas, como HPC Challenge [53] que consta de siete *benchmarks* diferentes. En cualquier caso, la lista refleja el significativo crecimiento de los sistemas de supercomputación de propósito general en los últimos treinta años.

En su 56ª edición, publicada en noviembre de 2020, el mínimo necesario para entrar en la lista subió hasta los 1,32 *petaflops* en el *benchmark* HPL (*High Performance Linpack*) [54], alcanzando conjuntamente los 500 sistemas un rendimiento total de 2,43 *exaflops* y una concurrencia media de 145.465 núcleos[1]. La primera posición continuó ocupada por el supercomputador Fugaku [55], con 442 *petaflops*. Asimismo, cabe destacar que este sistema obtuvo un rendimiento de 2,0 *exaflops* en el *benchmark* de precisión mixta HPC-AI [56], que combina HPC con Inteligencia Artificial, superando en 0,6 puntos su anterior registro. Estos datos representan las primeras medidas de referencia superiores al *exaflop* para cualquier precisión en cualquier tipo de *hardware* [50].

[1]Debido a la rápida evolución que experimentan los sistemas HPC registrados en el TOP 500, y dado que esta lista es actualizada semestralmente, existe la posibilidad de que los datos citados queden obsoletos a fecha de publicación de este trabajo.

2.1.1. Taxonomía de Flynn

Existen varias taxonomías para clasificar computadores como las propuestas por Feng [57], Händler [58] o Flynn [59]. Esta última clasifica las arquitecturas de computadores en función del número de instrucciones y de los flujos de datos que éstas pueden procesar simultáneamente, diferenciando así el paralelismo de datos del de tareas. Aunque actualmente presenta importantes limitaciones, se sigue utilizando como una primera aproximación para comprender y encuadrar las distintas infraestructuras de cómputo. En concreto, define las siguientes categorías:

- SISD (*Single Instruction Single Data*): ejecución secuencial con un solo procesador y un único hilo de ejecución que no emplea paralelismo en instrucciones ni datos. Se trata del concepto de la arquitectura de Von Neumann.
- MISD (*Multiple Instruction Single Data*): varias instrucciones operan en un único flujo de datos. No es una arquitectura habitual, aunque es utilizada, por ejemplo, para ejecutar pruebas de tolerancia a fallos o realizar búsquedas de patrones [60].
- SIMD (*Single Instruction Multiple Data*): una única instrucción se aplica al mismo tiempo sobre múltiples conjuntos de datos en un único procesador o en múltiples elementos de cómputo. Era la infraestructura básica de los procesadores vectoriales y actualmente forma parte de instrucciones y estructuras de control más complejas en microprocesadores y supercomputadores heterogéneos [33] o en coprocesadores como Intel Xeon Phi y unidades de procesamiento gráfico (GPU, *Graphics Processing Units*). Estas últimas han aumentado significativamente su importancia en la computación de alto rendimiento debido a su bajo coste en relación a su potencia computacional y gran ancho de banda de memoria, siendo muy adecuadas por ejemplo para cálculos de matrices densas [61].
- MIMD (*Multiple Instruction Multiple Data*): es la arquitectura paralela más utilizada actualmente aunque presenta varias subclases. En ella múltiples computadores o procesadores (cores) autónomos ejecutan distintas instrucciones con diferentes datos de forma independiente [62].

Los modelos de programación SPMD (*Single Program Multiple Data*) y MPMD (*Multiple Program Multiple Data*) en los que uno o varios programas independientes se ejecutan en varios procesadores de forma simultánea, son considerados subcategorías de la clasificación MIMD [33], detallada con mayor profundidad en el siguiente apartado.

2.1.2. Computadores MIMD (*Multiple Instruction Multiple Data*)

Independientemente de la taxonomía de Flynn, la forma más común de clasificar los computadores MIMD es mediante la disposición de su memoria, considerando ésta como compartida, distribuida o híbrida, tal y como se describe a continuación:

- Memoria compartida (multiprocesadores): arquitecturas fuertemente acopladas donde varios procesadores acceden de forma simultánea a un espacio de direcciones físicas de memoria principal común y compartido, permitiendo la ejecución simultánea de varios hilos correspondientes a uno o más procesos. En consecuencia, los cambios aplicados sobre un dato determinado pueden afectar a otros procesadores. La red que interconecta los procesadores con los bancos de memoria resulta clave. Asimismo, puesto que puede darse la existencia de varias copias de la misma línea de caché en distintos procesadores del sistema, la consistencia entre los datos almacenados en caché y memoria debe ser asegurada por el correspondiente protocolo de coherencia. Los computadores de memoria compartida se clasifican en:

  - UMA (*Uniform Memory Access*): modelo de memoria "plano" con igual latencia y ancho de banda para todos los procesadores y todas las ubicaciones de memoria, de manera que éstos emplean la misma cantidad de tiempo en acceder a cada bloque. Se trata de un multiprocesamiento simétrico (SMP, *Symmetric Multi-Processing*) [63]. La mayoría de sistemas multiprocesador actuales utilizan esta arquitectura, siendo empleada en servidores, ordenadores de escritorio y portátiles, o también, como parte de MPP (*Massively Parallel Processors*) mucho más grandes [33]. En el caso de los procesadores multinúcleo (*multicore*), la arquitectura se aplica a los núcleos, tratándolos como procesadores separados.
  - NUMA (*Non-Uniform Memory Access*): sistemas DSM (*Distributed Shared Memory*) donde la memoria se distribuye físicamente pero se comparte de forma lógica, de modo que en total se utiliza un único espacio de direcciones [63] aunque cada procesador disponga de su propia memoria local. De este modo, el rendimiento varía en función de cada acceso, según la distancia a la memoria utilizada. Los microprocesadores modernos posibilitan la utilización de canales de comunicación de memoria local de alta velocidad, mientras que las redes de conexión externas (más lentas) posibilitan el acceso a toda la memoria [33]. En general, los multiprocesadores NUMA se clasifican en *cache coherence* NUMA (ccNUMA) o *non-coherence* NUMA, en función de la integridad de los datos almacenados en las memorias caché locales.

- Memoria distribuida (multicomputadores): arquitecturas débilmente acopladas donde cada procesador posee su propia memoria física, la cual no es visible ni directamente accesible para el resto de procesadores. Es necesaria la comunicación entre ellos para compartir datos, pero ésta se realiza a través de la red en lugar de utilizar buses internos, por lo que para trabajar conjuntamente deben enviarse mensajes entre procesos. En general, se consideran principalmente arquitecturas MPP y clústeres, predominando actualmente estas últimas [50]. A continuación se detallan los principales tipos de entornos de memoria distribuida existentes:

  - MPP: grupos separados de nodos de procesamiento, cada uno con su propia memoria local no compartida, interconectados entre sí a través de una red de altas prestaciones, con baja latencia y elevado ancho de banda. Esta arquitectura persigue simplificar el diseño y eliminar ineficiencias que dificultan la escalabilidad [33].

  - Clúster: formados por multitud de nodos de cómputo totalmente autónomos e independientes, y en muchos casos heterogéneos, integrados mediante una red privada de alta velocidad, cada uno con su propio sistema operativo [64]. Sus nodos son más básicos y suelen alcanzar un rendimiento menor en comparación con procesadores paralelos masivos, pero en conjunto presentan una mejor relación coste-beneficio y una escalabilidad ampliamente superior respecto a computadores de igual potencia o disponibilidad.

- Computadores híbridos: generalmente estructurados de forma jerárquica y compuestos por conjuntos de procesadores interconectados que combinan las características de ambos enfoques de memoria compartida y distribuida. En general, tanto los sistemas de procesamiento como los de memoria están organizados en capas, cada una con características específicas. De este modo, la caché es dividida en niveles, unos dedicados a núcleos concretos y otros compartidos por todos los núcleos de ejecución. A menudo están formados por computadores SMP interconectados, que permiten utilizar conjuntamente modelos de programación paralela de memoria compartida y distribuida, aprovechando las ventajas de ambas (ver apartado 2.3 sobre modelos de programación paralela), aunque a cambio aumente la complejidad de su uso.

2.1.3. Redes y tecnologías de interconexión

La red que conecta las distintas unidades de procesamiento y los nodos de cómputo presenta gran importancia, especialmente debido a que el *overhead*[2] de las comunicaciones impacta de forma significativa en el rendimiento.

La topología de red más utilizada en los centros HPC es Fat tree [65], aunque existe una amplia variedad, principalmente basadas en 2D Torus [66, 67], De Bruijn [68] y Butterfly [69], diseñadas para grandes anchos de bancha, baja latencia y balanceo en la red de comunicaciones. Por otro lado, los centros de datos también emplean topologías como DCell [70], Bcube [71] y SCautz [72], más orientadas a una alta escalabilidad [73]. Del mismo modo, se han propuesto muchas topologías de bajo diámetro como Slim Fly [74], Dragonfly [75] o Jellyfish [76] que reducen significativamente los costes, el consumo energético y la latencia [77].

En relación a las tecnologías de interconexión de red, en la actualidad predomina Infiniband [78, 79] frente a Omni-Path [80] y Gigabit Ethernet [81]. En concreto, en la última actualización del Top 500 (noviembre de 2020) destaca Gigabit Ethernet con un 50,8 % (17,4 % 10G Ethernet, 16 % 25G, 14,6 % 40G y tan solo un 2,8 % 100G Ethernet), mientras que Infiniband está presente en el 31 % de los sistemas, Omni-Path en el 9,4 %, Myrinet [82] en el 0,2 %, y un 8,6 % se corresponde con tecnologías específicas de cada fabricante. Sin embargo, es importante destacar que siete de las diez primeras entradas del ranking utilizan Infiniband, mientras que la primera aparición de Gigabit Ethernet se

[2]Trabajo o tiempo adicional al requerido realmente para realizar la tarea de cómputo. Es originado por la gestión de los recursos y la planificación de tareas, las sincronizaciones paralelas y las comunicaciones entre procesos e hilos de ejecución, así como por otras funciones complementarias a las operaciones estrictamente computacionales [33].

encuentra en la posición 67ª, lo cual refleja la clara tendencia actual de Infiniband frente al resto de tecnologías.

Los clústers más básicos emplean habitualmente una única red Gigabit Ethernet para el tráfico de gestión, el intercambio de datos y los cálculos computacionales. En caso de ser necesario mejorar el ancho de banda o la latencia, es común emplear una única interconexión de alta velocidad como Infiniband u Omni-Path. Por contra, los clústeres más avanzados suelen implementar tres redes, una dedicada al tráfico computacional y el almacenamiento e intercambio de datos, una segunda destinada a la gestión y los inicios de sesión, y la última dedicada al mantenimiento o configuración de los nodos. De esta forma se asegura que el tráfico computacional no se vea afectado por el tráfico de control necesario para el correcto funcionamiento de los nodos.

2.1.4. *Slurm*

Los sistemas de computación de alto rendimiento ofrecen acceso a muchos usuarios de forma simultánea, por lo que requieren una adecuada gestión y distribución de los recursos disponibles, necesitando una herramienta que se encargue de balancear sus cargas en relación al envío y la ejecución de tareas. Entre los gestores de trabajo más utilizados se encuentran Slurm [83], LSF (*Load Sharing Facility*) [84], PBSPro [85], Maui [86], Moab [87], Torque [88] y Sun Grid Engine [89]. A continuación se detalla el primero de ellos, por ser el utilizado en el centro de supercomputación donde se ha desarrollado esta investigación, y emplearse en el transcompilador para ejecutar los programas paralelizados.

Slurm (*Simple Linux Utility for Resources Management*) es un sistema de planificación de trabajos y gestor de colas de código abierto, para el acceso a recursos de cómputo y la planificación de tareas en clústeres Linux, usado ampliamente en centros de procesamiento de datos, universidades y empresas de todo el mundo, siendo además empleado en múltiples supercomputadores de ámbito internacional [50].

Este sistema de gestión se encarga de asignar a los usuarios los distintos recursos computacionales disponibles, proporciona un *framework* para iniciar, ejecutar y monitorizar el trabajo en el conjunto de nodos asignados, y arbitra la disputa por los recursos gestionando colas de trabajo pendiente. Sus componentes básicos son el proceso `slurmctld`, ubicado en el nodo principal del clúster, un proceso en segundo plano `slurmd` ejecutado en cada nodo de cómputo y una serie de comandos de usuario, entre los cuales destacan especialmente: `srun`, para asignar recursos de cómputo a una tarea de ejecución; `squeue`, que muestra por consola información sobre el estado actual de cada cola y ofrece la lista de las tareas programadas y en ejecución; `scancel`, utilizado para abortar la ejecución de trabajos en funcionamiento; y `sbatch`, empleado para enviar al gestor el *shell script* a ejecutar. La diferencia principal de este último comando frente a `srun`, es que permite ubicar un conjunto completo de programas mediante un único fichero [62].

En el Código 2.1 se muestra un *script* de ejemplo que utiliza diferentes directivas de `sbatch`: `ntasks`, número de cores a utilizar; `job-name`, nombre asignado al trabajo; `mail-user`, dirección de correo a donde enviar las notificaciones relacionadas con el

Código 2.1: *Script* de ejemplo de Slurm.

```
#!/bin/bash
#SBATCH --ntasks=48
#SBATCH --job-name=hello_world
#SBATCH --mail-user=user@mail.es
#SBATCH --mail-type=ALL
#SBATCH --output=hello_world.out
#SBATCH --error=hello_world.err
#SBATCH --partition=lusitania
#SBATCH --time=0-00:01:00
srun -n $SLURM_NTASKS$ hello_world
```

trabajo; `mail-type`, define los eventos que serán informados al usuario; `output`, fichero de salida estándar; `partition`, partición a la que enviar el trabajo; y `time`, límite de tiempo para realizar la ejecución en formato `D-HH:MM:SS`. Adicionalmente, en el ejemplo se emplea la variable de entorno `$SLURM_NTASKS$` que señala el número total de cores por trabajo .

2.2. Conceptos fundamentales sobre paralelización

2.2.1. Concurrencia y paralelismo

La concurrencia consiste en la ejecución de dos o más tareas de cómputo independientes en un mismo periodo de tiempo, lo cual no implica que se ejecuten a la vez. Es empleada por ejemplo, en sistemas multitarea (*multitasking*) con un único núcleo que aplican *time-slicing* como paralelismo virtual. El paralelismo real, sin embargo, permite ejecutar esas tareas en diferentes recursos *hardware* o elementos de procesamiento y literalmente al mismo tiempo, con el objetivo de aumentar el rendimiento global, usándose por ejemplo en procesadores multinúcleo. Así, aunque los programas paralelos cumplen la definición de concurrencia, un proceso concurrente no tiene por qué ser paralelo.

2.2.2. Paralelización de programas

El principal problema que afronta la paralelización de códigos secuenciales es la existencia de dependencias de datos en el código, de manera que dos o más instrucciones operan sobre la misma ubicación de memoria e impiden su ejecución en paralelo. La literatura recoge distintas estrategias para el análisis y diseño de este tipo de algoritmos, considerando la descomposición y asignación de tareas a las unidades de procesamiento. Entre ellas destaca especialmente la metodología PCAM (*Partitioning, Communication, Agglomeration, and Mapping*) [90], que presenta dos descomposiciones iniciales para comenzar a paralelizar el problema: del dominio y funcional. La descomposición del dominio consiste en dividir los datos a procesar (por ejemplo mediante reducción,

convirtiendo una matriz en varias submatrices), mientras que la funcional se centra en realizar una descomposición natural del problema en base a objetivos computacionales independientes [91]. En general, la paralelización de códigos secuenciales busca maximizar la computación y minimizar las comunicaciones entre procesos, requiriendo una serie de pasos, independientes del modelo de programación utilizado, con objeto de disminuir de la forma más eficiente posible los tiempos de ejecución. Para ello es necesario considerar las dependencias de datos y de control existentes en cada código. A continuación se describen estos pasos, que consideran: la descomposición en tareas y su granularidad, la planificación o asignación de las cargas de trabajo, y su mapeo o distribución entre los procesadores o núcleos de ejecución.

En primer lugar, las distintas operaciones de cómputo son descompuestas en tareas, determinando las dependencias existentes entre ellas. El objetivo es que cada una, que puede estar comprendida por una o varias instrucciones, sea ejecutada en un procesador distinto, de modo que se reduzca lo máximo posible el tiempo de ejecución. La descomposición puede realizarse de forma estática, identificando todas las tareas antes de que el algoritmo comience a ejecutarse, o dinámica, en la cual las tareas paralelas pueden variar en tiempo de ejecución. En esta última hay que considerar igualmente el correcto equilibrio de la carga durante los distintos cambios.

Asimismo, la descomposición de tareas debe efectuarse teniendo en cuenta que la ejecución en paralelo de cualquier fragmento de código siempre requiere un tiempo adicional para coordinar las tareas asociadas (tal y como se expone en el apartado 2.2.4 sobre las medidas de rendimiento de los programas paralelos). Las tareas deben tener suficiente carga computacional, de modo que el tiempo de ejecución paralelo compense el requerido en la planificación y distribución del trabajo. Adicionalmente, la granularidad es determinada por el número y el tamaño de las tareas en las que se descompone un problema. Una granularidad gruesa (*coarse-grained granularity*) implica menos comunicación entre las distintas tareas y un menor nivel de paralelismo, estando asociada generalmente a un balanceo de carga más pobre y una mayor localidad en los accesos a los datos. La granularidad fina (*fine-grained*) por el contrario, descompone el problema en un mayor número de tareas pequeñas, existiendo usualmente un mejor equilibrio de carga a costa de una menor localidad y un mayor tiempo de comunicación entre las tareas. Una programación paralela óptima requiere encontrar un equilibrio perfecto entre el número de tareas paralelas ejecutadas y su granularidad.

La planificación (*scheduling*) paralela o asignación a procesos o hilos de ejecución debe realizarse buscando un correcto equilibrio de la carga de trabajo destinada a cada uno, para lo cual hay que tener en cuenta distintos factores, como el número de operaciones de cómputo a realizar, los accesos a memoria o las comunicaciones entre los procesos o hilos. De forma similar a la descomposición de tareas, la planificación puede ser estática (en tiempo de compilación) o dinámica (en tiempo de ejecución). En el apartado 2.3.3, dedicado a OpenMP, se profundiza en los distintos tipos de planificaciones y sus características en este modelo de programación de memoria compartida.

La distribución (mapeo) de hilos a las unidades de ejecución (cores físicos) disponibles, es realizada por el sistema operativo y puede ser dirigida mediante distintas declaraciones en el propio programa. Aunque es posible mapear varios hilos a un único core en el caso

de que no haya unidades de ejecución suficientes, el objetivo es alcanzar un equilibrio de carga entre las distintas unidades de ejecución, así como un número de comunicaciones entre ellas que asegure el mejor rendimiento.

2.2.3. Niveles de paralelismo

Existen distintos niveles que influyen en la granularidad de las tareas resultantes y que son principalmente considerados para explotar el paralelismo (ordenados de granularidad más fina a más gruesa): de instrucción, de datos, de bucles y de tareas o funcional.

La posibilidad de explotar el paralelismo a nivel de instrucción depende de que no haya dependencias entre ellas. Considerando dos instrucciones del mismo flujo, I_1 e I_2, y los registros en memoria R_1 a R_5, se definen las siguientes dependencias posibles (ver Tabla 2.1 [92]):

- Dependencia de flujo o verdadera (*flow/true dependency*): representa el caso en el que I_1 almacena su resultado en una variable o registro que posteriormente es utilizado como operando por I_2. Generalmente esta dependencia no puede ser resuelta e impide que ambas instrucciones sean ejecutadas en paralelo. En el ejemplo de la tabla, la instrucción I_2 podría utilizar un valor anterior del registro R_1 para realizar su operación, provocando un error en el resultado.
- Antidependencia (*antidependence*): generada de forma inversa a la anterior cuando I_1 emplea en sus operandos una localización de memoria usada posteriormente por I_2 para almacenar el resultado de sus operaciones. Respecto al ejemplo de la tabla, una ejecución previa de I_2 provocaría que I_1 usase un valor incorrecto del registro R_2, escrito anteriormente por I_2. Como se detalla más adelante, en el caso de que ambas instrucciones estuviesen contenidas en un bucle, la dependencia podría ser eliminada utilizando en cada iteración una copia privada de la variable.
- Dependencia de salida (*output dependence*): producida cuando I_1 e I_2 escriben sus resultados en el mismo registro o variable. En el ejemplo mostrado en la tabla, invertir el orden de ambas instrucciones supondría que las instrucciones posteriores utilizarían un valor incorrecto de R_1. En un bucle, la paralelización sería posible siempre que la ubicación de memoria de R_1 fuese privada para ambas instrucciones.

Por otro lado, el paralelismo de datos se basa en la aplicación de operaciones independientes sobre diferentes partes de una misma estructura de datos. Distintos elementos de éstas (generalmente vectores) son repartidos entre los procesadores disponibles, que aplican las instrucciones sobre cada fragmento. Si no existen

Tabla 2.1: Tipos de dependencias entre instrucciones.

Dependencia de flujo	Antidependencia	Dependencia de salida
$I_1 : \mathbf{R_1} \leftarrow R_2 + R_3$	$I_1 : R_1 \leftarrow \mathbf{R_2} + R_3$	$I_1 : \mathbf{R_1} \leftarrow R_2 + R_3$
$I_2 : R_5 \leftarrow \mathbf{R_1} + R_4$	$I_2 : \mathbf{R_2} \leftarrow R_4 + R_5$	$I_2 : \mathbf{R_1} \leftarrow R_2 + R_3$

dependencias, las operaciones pueden ser ejecutadas simultáneamente por varios procesadores de un sistema paralelo o distribuido, requiriendo el intercambio de mensajes entre ellos en caso necesario.

En relación al paralelismo aplicado en los bucles, estos permiten realizar cálculos recorriendo iterativamente estructuras de datos. Para poder ser paralelizados es necesario que no existan dependencias entre instrucciones de distintas iteraciones (*loop-carried dependencies* o dependencias acarreadas) y éstas puedan ser ejecutadas sin un orden concreto (*out-of-order executions*). En este sentido, las dependencias identificadas a nivel de instrucción afectan al bucle cuando I_1 e I_2 dependen explícita o implícitamente de diferentes valores de la variable de control del mismo. Cuando las condiciones de paralelización se cumplen, es posible distribuir sus iteraciones entre los núcleos de procesamiento disponibles.

En caso necesario, las condiciones pueden obtenerse, por ejemplo, cambiando previamente el ámbito de la variable mediante la correspondiente directiva de paralelización, transformando el código fuente o aplicando ambas soluciones. En este sentido, las antidependencias pueden ser eliminadas mediante privatización de la variable implicada y ofreciendo a cada hilo o iteración una copia de la misma, de esta forma, los datos pueden ser leídos antes de que otra instrucción los modifique.

Por contra, en el caso de las dependencias de flujo, suele ser necesario alterar el algoritmo para poder explotar su paralelismo. Para ello existen distintas técnicas, como paralelización parcial (*partial parallelization*), refactorización (*refactoring*), fisión (*fissioning*) o *loop skewing*, entre otras [91]. En los casos más sencillos, las dependencias de flujo pueden ser eliminadas mediante reducciones (cuando la dependencia está causada por instrucciones de acumulación sobre una variable). En cualquier caso, si no es factible eliminar una dependencia, el código debe ser reescrito buscando la mayor paralelización posible.

Adicionalmente, se considera que un programa puede aprovechar el paralelismo de tareas (también denominado paralelismo funcional) cuando contiene diferentes partes independientes entre sí que pueden ser ejecutadas de forma paralela [93]

2.2.4. Medidas de rendimiento de programas paralelos

Entre los factores que limitan la ejecución paralela destacan principalmente los siguientes:

- Limitaciones algorítmicas: en ocasiones ciertas partes de un código no pueden ser paralelizadas, por ejemplo debido a las dependencias descritas en el anterior apartado, que hacen que las instrucciones tengan que ser ejecutadas de forma secuencial o en un orden determinado.
- Cuellos de botella: limitan la concurrencia y son provocados habitualmente por la propia arquitectura e infraestructura física utilizadas.

- *Overhead*: la ejecución en paralelo de un programa secuencial requiere siempre un *overhead* o tiempo adicional para la coordinación de las tareas paralelas (crear, iniciar y detener hilos y determinar qué trabajo debe realizar cada una) así como para las esperas en barreras, secciones críticas y cerrojos. Este *overhead* es, por tanto, inherente a cada fragmento de código que se ejecute de forma paralela.
- La comunicación entre diferentes partes de un sistema paralelo suele implicar cierta serialización de las ejecuciones.

A continuación se describen tres conceptos fundamentales para medir el rendimiento de los programas paralelos: *speedup*, eficiencia y escalabilidad.

2.2.4.1. *Speedup*

El *speedup* o aceleración de un programa (S_p), representa la mejora obtenida en su tiempo de ejecución al ser computado de forma paralela. Es definido como el cociente entre el tiempo de ejecución secuencial[3] (T_s) y su tiempo de ejecución paralelo (T_p), siendo p el número de unidades de cómputo (procesadores o núcleos) empleadas:

$$S_p = \frac{T_s}{T_p} \tag{2.1}$$

En el mejor de los casos, el tiempo de ejecución de un programa paralelo utilizando p procesadores será p veces inferior al de su ejecución en un único procesador (teniendo en cuenta que todos presenten idénticas características). Normalmente se cumple que $1 \leq S_p \leq p$, de forma que el valor máximo que puede alcanzar el *speedup* de un algoritmo paralelo es p.

Un código presenta *speedup* lineal cuando éste converge hacia p al aumentar tanto el tamaño del problema como el número de procesadores, lo cual sucede en ciertos algoritmos, como en las simulaciones de Monte Carlo [94], donde los procesadores no necesitan comunicarse entre sí. Por otro lado, un *speedup* es calificado como súper lineal cuando $S_p > p$ al crecer el tamaño del problema y los recursos de cómputo. Esto sucede por ejemplo, en algoritmos de búsqueda basados en *backtracking* o cuando en la versión secuencial se producen más fallos en caché. Por último, un algoritmo paralelo presenta *speedup* sublineal cuando $S_p < p$ [95]. De hecho, es habitual que el tiempo se vea reducido en un orden menor a p, debido al *overhead* causado por las sincronizaciones y a las dependencias existentes entre los datos.

La ley de Amdahl [96, 97], basada en problemas o cargas de trabajo de tamaño fijo, establece que la mejora alcanzada en el rendimiento de un sistema paralelo está limitada por la fracción de tiempo que éste se ejecuta de forma secuencial. En primer lugar, se considera:

$$f_s + f_p = 1 \tag{2.2}$$

[3]Tiempo obtenido al ejecutar secuencialmente el código completo en un único procesador. En general, se considera cualquier versión secuencial aunque debería tenerse en cuenta el tiempo de ejecución de la mejor implementación [92].

siendo f_s la fracción secuencial y f_p la paralela. De este modo, el tiempo de ejecución paralelo, T_p, es definido por:

$$T_p = T_s\left(f_s + \frac{f_p}{p}\right) \tag{2.3}$$

Por tanto, se determina que el *speedup* puede ser definido como:

$$S_p = \frac{1}{f_s + \frac{1-f_s}{p}} \tag{2.4}$$

En términos generales, la Ley de Amdahl denota que es el algoritmo el que determina la mejora de velocidad, no el número de procesadores o núcleos de ejecución. En consecuencia, el incremento de velocidad de un programa utilizando múltiples unidades de procesamiento está fuertemente limitada por la fracción secuencial del mismo [92].

La ley de Gustafson [98], también conocida como ley de Gustafson-Barsis, establece que cualquier problema suficientemente grande puede ser eficientemente paralelizado. Es aplicada sobre problemas escalables, donde el tamaño del problema se incrementa al aumentar el tamaño de la máquina o se dispone de un tiempo fijo para realizar una determinada tarea. La ley define el *speedup* como:

$$S_p = p + f_s \cdot (1-p) \tag{2.5}$$

Gustafson ofrece así un punto de vista distinto sobre el procesamiento paralelo. A diferencia de la ley de Amdahl, no propone una solución a un problema con un tamaño determinado y limitado, sino que aprovecha la parte paralelizable para alcanzar así un *speedup* significativo y no acotado excepto por los propios sistemas de cómputo en los que se ejecuta el código.

2.2.4.2. Eficiencia

La eficiencia (E) determina el grado de utilización de los procesadores o núcleos para la resolución del problema. Es definida por el cociente entre el *speedup* y el número de unidades de procesamiento:

$$E = \frac{S_p}{p} \tag{2.6}$$

El valor máximo que puede alcanzar es 1, que denota un 100 % de aprovechamiento. Del mismo modo, un *speedup* ideal ($S_p = p$) se corresponde con una eficiencia $E = 1$.

2.2.4.3. Escalabilidad

La escalabilidad puede ser definida como la capacidad de un determinado algoritmo para mantener sus prestaciones cuando aumenta el número de unidades de proceso y el tamaño del problema en la misma proporción. Es decir, representa la capacidad de utilizar de forma efectiva un incremento en los recursos computacionales. Un algoritmo paralelo escalable mantiene su eficiencia constante al aumentar el tamaño del problema. En caso

contrario, pese a incrementar el número de unidades de procesamiento, no se conseguirá mejorar la eficiencia aunque también crezca el tamaño del problema, de forma que cada vez se irá aprovechando menos la potencia de cómputo.

Asimismo, la escalabilidad depende de varias propiedades del algoritmo y su ejecución paralela. Es habitual que para un problema de tamaño fijo, exista una saturación del *speedup* a medida que aumenta el número de procesadores, provocando que la eficiencia comience a disminuir de forma más o menos acentuada.

2.3. Modelos de programación paralela

La literatura ofrece diversas perspectivas para afrontar los desafíos de programación planteados por la computación paralela, principalmente: librerías para la escritura de programas paralelos, interfaces con directivas de paralelización, lenguajes de programación paralela, y compiladores de paralelización automática [19].

Adicionalmente, los modelos de programación paralela también se clasifican según el sistema de memoria (compartida o distribuida) utilizado. De este modo, en los de memoria distribuida, entre los cuales destaca MPI (*Message Passing Interface*, Interfaz de Paso de Mensajes) [99, 100], los procesos se ejecutan con espacios de direcciones separados y se comunican mediante paso de mensajes entre ellos. En los de memoria compartida, como OpenMP, el programa es dividido en varios procesos (hilos de ejecución) que intercambian datos relacionados con su espacio de direcciones principalmente a través de variables compartidas.

En relación a las librerías para la escritura de códigos paralelos, existen diferentes enfoques de programación, como MPI y CUDA (*Compute Unified Device Architecture*, arquitectura unificada de dispositivos de cómputo), los cuales además pueden ser combinados para alcanzar mejores resultados mediante programación híbrida, utilizando MPI para distribuir el cómputo y CUDA para las ejecuciones principales [10, 101, 102].

Además se dispone de APIs (*Application Programming Interfaces*) como OpenMP, OpenACC (*for Open Accelerators*) [11] y OpenHMPP [103], que están diseñadas específicamente para arquitecturas heterogéneas y realizan la paralelización mediante la inserción de directivas en programas secuenciales con lenguajes C, C++ y Fortran o aplicaciones GPUs.

También existen modelos híbridos que emplean paradigmas de memoria compartida para la paralelización dentro de cada nodo y paso de mensajes para la intercomunicación entre ellos. Aunque estos carecen de las ventajas que ofrecen otros enfoques como MapReduce [104] o Spark [105], también alcanzan niveles de rendimiento adecuados [106, 107].

Además, existen lenguajes de programación paralelos, entre los que destacan especialmente los PGAS (*Partitioned Global Address Space*) [12] como Chapel [108], Fortran y C++ Coarrays [109, 110], X10 [111] y UPC [112], que permiten gestionar

datos y comunicaciones en sistemas de memoria distribuida. Están igualmente basados en lenguajes como C, C++ y Fortran, y se centran en preservar un espacio de direcciones compartido y global, particionado para distinguir de forma lógica entre accesos locales y remotos, optimizando así el rendimiento en arquitecturas paralelas de gran escala [12].

También, existen compiladores para la paralelización automática de programas secuenciales que determinan qué partes pueden ejecutarse de forma concurrente y generar el código paralelo correspondiente. A continuación se detallan este tipo de aproximaciones.

2.3.1. Modelo poliédrico

La mayoría de los compiladores de paralelización de códigos propuestos en la literatura están basados en análisis de dependencias entre instrucciones y modelos poliédricos [113], apoyándose en representaciones intermedias que emplean, por ejemplo, árboles de sintaxis abstracta (AST, *Abstract Syntax Tree*) y gráficos de dependencia de datos (DDG, *Data Dependence Graph*) o de control de flujo (CFG, *Control-Flow Graph*). Sin embargo, logran un éxito limitado en la paralelización automática. Por ello, ésta sigue suponiendo un desafío para la comunidad HPC.

En general, la mayoría de las técnicas presentadas en la literatura son enfoques parciales para la paralelización automática o modelan bucles simples de forma individual. Sin embargo, muchos compiladores no consiguen aún paralelizar ciertos programas secuenciales, principalmente cuando estos se enfrentan a determinadas variaciones sintácticas, como el uso de punteros y flujos de control complejos [19].

El modelo poliédrico, también conocido como politopo, (*polytope*), es utilizado por compiladores como GCC (*GNU Compiler Collection*) [13], LLVM [114, 115], Pluto [116, 117], PPCG [118] e IBM XL [32]. Consiste en una representación algebraica de programas secuenciales que combina análisis y transformaciones de alto nivel con el objetivo de poder diseñar heurísticas de optimización [119]. En concreto, permite capturar el comportamiento de un programa mediante funciones relacionadas, a partir de las cuales realizar análisis complejos y optimizar el código mediante transformaciones. Las partes representadas mediante el modelo poliédrico se denominan SCoPs (*Static Control Part*) y consisten en vectores con varios elementos sobre cada operación a analizar, que constituyen un conjunto de inecuaciones que representan el dominio de iteración, las funciones de acceso, la planificación y las dependencias entre sentencias. Estas inecuaciones, al ser resueltas, definen un espacio que representa el conjunto de transformaciones del código que preservan la semántica original del programa, es decir, que respetan las dependencias entre instrucciones.

Sin embargo, a pesar de sus ventajas, generalmente solo es posible utilizarlo en bucles anidados cuyo comportamiento lineal pueda ser determinado en tiempo de compilación. Esto hace que compiladores que ofrecen paralelización automática basada en representación poliédrica, como GCC, que utiliza para ello el *framework* Graphite [120], sea ineficaz en aplicaciones a gran escala y en asignaciones y reducciones irregulares que implican referencias indirectas a memoria [19]. En general, solo los bucles anidados con límites afines y las expresiones condicionales pueden ser representados de forma

poliédrica. Así, bucles con acceso indirecto a memoria o punteros, no pueden ser paralelizados con estas técnicas, aunque existen propuestas que persiguen eliminar estas limitaciones [119]. De hecho, en las últimas décadas la comunidad investigadora ha ampliado la teoría, las herramientas y las bibliotecas poliédricas disponibles [121–124] aplicando el modelo en una extensa variedad de aplicaciones.

2.3.2. Paralelización automática

A continuación se recogen las propuestas más significativas recogidas en la literatura relacionadas con la paralelización automática de códigos secuenciales.

Existen *frameworks* poliédricos como AlphaZ [125] y CHiLL [126] que, aunque no son automáticos ni están diseñados para sistemas distribuidos [127], permiten a los usuarios explorar el espacio de posibles transformaciones.

Pluto [20] también utiliza el modelo poliédrico para paralelizar programas en C y Fortran, traduciendo previamente los bucles a una representación intermedia CLooG (*Chunky Loop Generator*) [128, 129]. Está especialmente diseñado para programas relacionados con álgebra y programación lineal, así como transformaciones de alto nivel. Sin embargo, solo funciona en bucles individuales que deben ser marcados previamente en el código fuente mediante directivas *pragma*[4] [130]. Además, al igual que sucede con GCC, es ineficaz con flujos de control complejos y operaciones irregulares que implican indirecciones[5] [19]. En general, Pluto no alcanza el nivel de rendimiento deseable debido a que sus técnicas de búsqueda de la paralelización consideran únicamente un subconjunto de todas las optimizaciones necesarias [127].

Existen otras propuestas basadas igualmente en los algoritmos de Pluto [131, 132], como Gpuloc [21] o “C to CUDA” [22]. Éste último fue el primer compilador automático poliédrico fuente a fuente para GPUs, pero permanece como prototipo y solo es aplicable a un pequeño conjunto de *benchmarks* [118].

Par4all [23, 24] es un compilador automático de paralelización y optimización para C y Fortran, que permite generar código OpenMP, OpenCL [133] y CUDA. Resulta eficiente en optimizaciones de bajo nivel [31] y, aunque no paraleliza bucles anidados y actúa únicamente sobre los bucles externos, incorpora heurísticas para paralelizar ciertos bucles internos. Utiliza el *framework* PIPS (*Parallelization Infrastructure for Parallel Systems*) para sus transformaciones, entre las que se incluyen la privatización de matrices, el reconocimiento de variables de reducción y la sustitución de variables de inducción. Aunque no está basado en el modelo poliédrico, utiliza interpretaciones abstractas que incluyen poliedros [118].

Cetus [25, 134, 135] es una herramienta de paralelización automática *source-to-source* que utiliza directivas OpenMP y emplea análisis estático para: implementar técnicas

[4] Las directivas *#pragma* permiten proporcionar información adicional al compilador, más allá de lo especificado por el propio lenguaje de programación.

[5] Referencias indirectas a los datos.

como la privatización escalar y matricial de variables; analizar dependencias de datos (principalmente mediante las desigualdades de Banerjee-Wolfe [136]); reconocer variables de reducción y sustituir variables de inducción. No soporta paralelismo anidado y paraleliza únicamente los bucles externos. Además, incorpora una interfaz gráfica de usuario.

Parallware [19] es un compilador comercial que admite distintas estrategias de paralelización basadas en el concepto de *kernel* independiente del dominio. Aplica técnicas de privatización y reconocimiento de variables de reducción, pero no realiza eliminaciones de dependencias mediante renombrado. De hecho, como se analiza en [137], Cetus no soporta ninguna de las transformaciones discutidas en [138] (fisión, fusión, desenrollado, *peeling* y sustitución de variables de inducción), mientras que Parallware solo soporta fusión. Además, aunque aborda la dependencia de datos con variables escalares, presenta dificultades en la paralelización de códigos con dependencias de iteración cruzada [137].

Los compiladores de Intel [14, 139], como ICC (*Intel C Compiler*), detectan automáticamente mediante heurística los bucles que pueden ser paralelizados. Para ello realizan análisis de flujos de datos e insertan directivas OpenMP, permitiendo la privatización de variables, la distribución de bucles y las permutaciones. Sin embargo, aunque algunos autores los sitúan entre los mejores paralelizadores [31], no ofrecen compilaciones *source-to-source* que permitan comparar el código secuencial con el paralelo, sino que se limitan a analizar los flujos de datos y realizar las transformaciones necesarias. Además, requieren conocer de antemano el número de iteraciones de cada bucle para poder paralelizarlo adecuadamente.

Rose [26, 140] es un compilador de código abierto que proporciona herramientas de análisis y transformación para aplicaciones C, C++ y Fortran. Sus optimizaciones incluyen autoparelización y desenrollado, *inlining*, bloqueo y fusión de bucles. Utiliza árboles AST, gráficos CFG de flujo de control y hace uso de tablas de símbolos. Utiliza la librería AutoPar [130, 141], que también ofrece paralelización automática en el compilador Clava [142], pero presenta limitaciones de memoria con la herramienta Petit [143], empleada para analizar las condiciones de carrera del código secuencial. Además, está basado en un *framework* concreto que requiere el aprendizaje previo de un lenguaje específico del dominio con palabras claves determinadas y una semántica propia, para describir el análisis y las transformaciones *source-to-source*.

Prema y Nasre ofrecen en [31] un completo análisis que compara de forma detallada Cetus, Par4all, Rose, ICC, y Pluto, identificando sus diferencias y debilidades, evaluando su rendimiento y sus técnicas de paralelización y demostrando que, a pesar de que estos y otros paralelizadores existentes en la literatura ofrecen considerables beneficios, aún presentan importantes problemas y rendimientos ineficientes. El estudio refleja asimismo la necesidad de analizadores, paralelización dirigida por el usuario, e incluso meta-auto-paralelizadores que puedan ofrecer los beneficios del uso conjunto de varios *frameworks*. Como conclusiones particulares, destacan las siguientes: Pluto funciona adecuadamente con programas muy concretos, pero falla a la hora de ofrecer un uso más general; los resultados de Cetus, Par4all y Rose dependen de las construcciones de cada programa y de sus posibilidades de paralelización; e ICC destaca por su vectorización, además de su paralelización y optimización adicional.

Prema presenta también en [144] un estudio más completo que tiene en cuenta a los paralelizadores de la publicación anterior y además incluye a Parallware, subrayando las fortalezas y debilidades de todos ellos, categorizándolos y analizándolos de forma detallada. Además, incluye varias tablas comparativas con información sobre los lenguajes utilizados, las arquitecturas soportadas y las estructuras y funciones admitidas. Entre otras particularidades, se destaca el hecho de que solamente Parallware e ICC admiten la paralelización de bucles *while*. Además, respecto a los bucles anidados, Pluto, Par4all, Parallware y Rose añaden las directivas paralelas únicamente en el bucle más externo, señalando que de esta manera se alcanza mayor eficiencia que haciéndolo en cada bucle. Respecto a las medidas de rendimiento, los experimentos realizados muestran que aunque Parallware obtiene los mejores resultados, el resto ofrece ventajas similares (salvo Rose, que requiere para ello la intervención manual del programador).

Traco [27] es un compilador para la paralelización de bucles basado en el *framework* ISSF [145] y en el analizador de dependencias Omega Calculator [146]. El preprocesador, desarrollado en Python, reconoce los bucles y los convierte al formato Petit aceptado por el analizador, que devuelve el conjunto de relaciones de dependencia del bucle. Posteriormente identifica las variables privadas y de reducción, estableciendo la paralelización del bucle mediante directivas OpenMP una vez que todas las relaciones de dependencia han sido eliminadas.

Existen también herramientas de paralelización automáticas relacionadas con compiladores específicos, como por ejemplo SUIF [28] o Polaris [29]. Este último requiere el uso del compilador KAP y únicamente paraleliza código fuente en Fortran77. Además, otros como LLVM Polly [30] tampoco ofrecen un rendimiento adecuado, debido a que solamente tienen en cuenta ciertas optimizaciones [127].

También es importante destacar que propuestas como Intel ICC no ofrecen transformaciones *source-to-source*, lo que evita que los usuarios puedan comparar el código original con el transformado, lo cual dificulta considerablemente tanto el aprendizaje como la mejora de las capacidades de programación paralela.

Algunos de los paralelizadores automáticos más destacables mantenidos actualmente (Cetus, ICC, Par4all, Pluto y Rose) presentan notables beneficios para el usuario, pero aún dependen de la intervención del programador y, en general presentan los siguientes problemas: deficiencias en la identificación y el uso de barreras implícitas; paralelizaciones ineficientes que producen *overhead*, *speedups* insuficientes e incrementos en los tiempos de ejecución; paralelizaciones de bucles demasiado pequeños que provocan disminuciones en el rendimiento; y problemas con los bucles anidados [31].

Así, aunque los enfoques existentes ofrecen importantes funcionalidades, aún no permiten reemplazar por completo las transformaciones manuales necesarias para el desarrollo de programas paralelos [31]. Además, salvo excepciones como Cetus, la mayoría no proveen interfaz gráfica y se centran únicamente en la línea de comandos, lo cual también dificulta el aprendizaje para usuarios inexpertos en el uso de la programación paralela. Por otro lado, las propuestas basadas en análisis estático (centrado en la paralelización del código fuente sin información adicional sobre la ejecución del programa), siguen presentando ventajas frente al análisis dinámico en tiempo de ejecución,

al ser generalmente más fáciles de utilizar ya que, por norma general y a diferencia de las herramientas dinámicas, no necesitan preparar previamente los programas secuenciales a paralelizar y suelen requerir menos tiempo de ejecución [130].

En resumen, la literatura recoge diversos enfoques y herramientas que utilizan representaciones intermedias y realizan análisis de dependencias, para ayudar en la paralelización de códigos secuenciales sin necesitar la interacción del usuario. Sin embargo, se centran principalmente en aumentar el *speedup* y reducir el tiempo de ejecución, sin considerar la formación de usuarios noveles en computación paralela. Consecuentemente, estos programadores presentan considerables dificultades para aprovechar sus ventajas y mejorar sus códigos, más allá de la propia paralelización de estos o de la ayuda automática de los compiladores. Por ello, las aportaciones de la investigación desarrollada se han centrado especialmente en estos usuarios.

Las características del transcompilador propuesto en esta tesis son descritas en profundidad en el Capítulo 3. Asimismo, sus limitaciones actuales y líneas de trabajo futuro pueden ser consultadas en el Capítulo 6. A continuación se resumen brevemente sus particularidades:

- La paralelización automática es realizada mediante la introducción de directivas OpenMP aplicadas sobre código C/C++, y se efectúa por medio del análisis estático del código y la identificación de dependencias. La utilización de OpenMP posibilita que el usuario pueda comparar ambos códigos en la versión paralelizada, facilitando su comprensión.
- El transcompilador se encuentra disponible en un archivo JAR (*Java ARchive*) ejecutable y portable[6] y presenta una interfaz gráfica, que además de mostrar el código secuencial y paralelo, destaca las líneas del programa que contienen las directivas de paralelización, para mayor comodidad del programador.
- Incorpora un módulo de optimización, basado en aprendizaje automático, que predice la planificación de ejecución paralela más adecuada para cada código (siendo descrito este componente en el apartado 3.2.4.2).
- Dispone de un sistema de apoyo que facilita el envío de los trabajos de cómputo y su análisis posterior (ver apartado 3.2.4.1).

2.3.3. OpenMP

OpenMP [9] es el modelo de programación de memoria compartida más utilizado en computación científica [63, 147]. Sus directivas posibilitan que el código original permanezca intacto y hacen posible que los programas resultantes puedan ser ejecutados tanto secuencialmente como de forma paralela, puesto que el archivo conserva al mismo tiempo ambas versiones. Esto ayuda además a potenciar el aprendizaje de aquellos usuarios con menos experiencia en este tipo de programación y simplifica notablemente

[6] Puede ejecutarse en cualquier sistema que cuente con Java Runtime Environment (JRE).

el desarrollo y la depuración del código. De hecho, los compiladores no compatibles con OpenMP ignoran directamente estas directivas, considerándolas simples comentarios.

Una de las ventajas más importantes de las computadoras con memoria compartida es que son más fáciles de programar y más rápidas cuando se comparten datos entre procesos o subprocesos. Además, el programador no tiene que lidiar con las comunicaciones para sincronizar todos los diferentes subprocesos que se ejecutan en los nodos. De este modo, OpenMP emplea una API más reducida que por ejemplo MPI (el estándar de facto para memoria distribuida) y su aprendizaje entraña también menor dificultad, siendo más sencillo para un usuario novel aprovechar todas sus características. De hecho, es mucho más complejo utilizar paso de mensajes que cualquier paradigma de memoria compartida [63], por lo que en general, la formación en programación paralela debería comenzar por este último modelo. Adicionalmente, en el caso de MPI se requiere que las estructuras de datos del programa sean divididas de forma explícita, generalmente debiendo paralelizar la aplicación como un todo para que el resultado sea adecuado, lo cual implica una mayor complejidad [41].

El conjunto de directivas de compilación ofrecidas por OpenMP consiste en comentarios o pragmas especialmente formateados (con sintaxis "`#pragma omp`"), que se insertan generalmente justo antes del código ejecutable a paralelizar [148]. De este modo, se planifican en tiempo de compilación un conjunto de hilos o subprocesos que ejecutarán una o varias instrucciones de forma paralela e independiente. Estos hilos también son denominados procesos ligeros (*lightweight processes*) [63], porque pueden compartir un espacio de direcciones común y acceder mutuamente a los datos. Se trata de un modelo *fork-join* en el cual el hilo maestro se ejecuta al comenzar el programa y se bifurca (*fork*) en el resto de hilos en cada una de las regiones paralelas determinadas por las correspondientes directivas (ver Figura 2.1). Posteriormente, estos hilos vuelven a unirse (*join*) al maestro tras completar sus respectivas ejecuciones. Es importante destacar que estas directivas no pueden ser aplicadas a todas las estructuras o bucles de un programa, debido a las dependencias entre instrucciones (ver apartado 2.2.3) por lo que el usuario debe seleccionar las partes del código que pueden ser paralelizadas.

La creación de hilos se implementa mediante la claúsula `omp parallel`. Además, existen distintos constructores que permiten especificar cómo distribuir las cargas de trabajo entre los hilos de ejecución. Así, posibilitan el reparto de las iteraciones de un bucle (`omp for`, `omp do`) o la asignación de bloques independientes de código a cada hilo (`sections`). También identifican ciertas instrucciones que deben ser ejecutadas únicamente por un hilo, empleando para ello una barrera de sincronización implícita al final del bloque (`single`), o ejecutan partes del código exclusivamente mediante el hilo maestro (`master`) [148].

OpenMP presenta además directivas de sincronización que permiten un control adicional de los hilos: con `critical section` se especifica que el bloque de código será ejecutado por un único hilo simultáneamente[7]; `atomic` identifica instrucciones atómicas; con `ordered` es posible ejecutar un bloque en el mismo orden en que se ejecutarían las

[7]Una región crítica asegura que un fragmento de código pueda ser ejecutado como máximo por un hilo al mismo tiempo. Si un hilo está procesando código dentro de una región crítica y otro intenta acceder, este último debe esperar (bloqueándose) hasta que el primero haya abandonado la región [148].

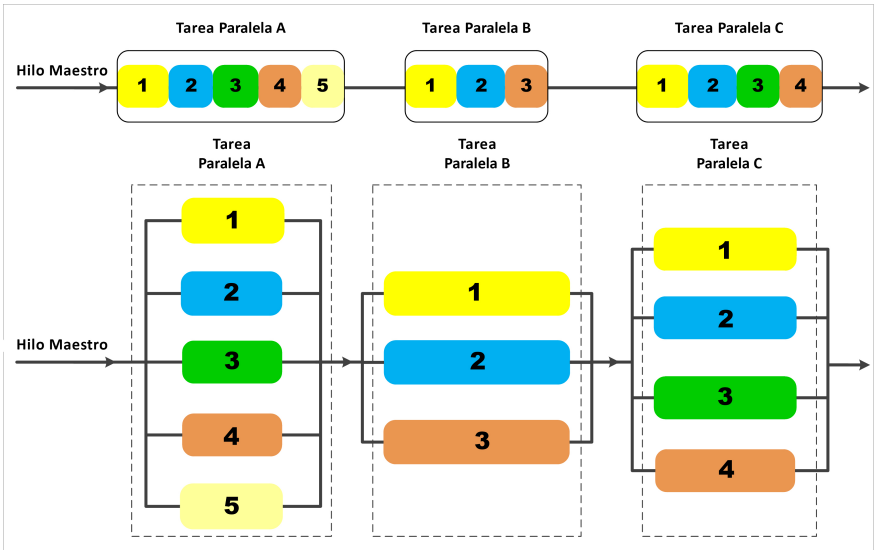

Figura 2.1: Modelo *fork-join* de OpenMP.

iteraciones de forma secuencial; mediante el uso de `barrier`, cada hilo espera hasta que el resto de hilos de un grupo alcancen dicha instrucción; y con `nowait` se controla que los hilos que hayan completado su trabajo puedan continuar, sin tener que esperar a que el resto de hilos del grupo finalicen. La API también proporciona rutinas con funciones similares al uso de semáforos, que ofrecen una mayor flexibilidad para las tareas de sincronización que la definición de secciones críticas o instrucciones atómicas.

A continuación se muestra la sintaxis general utilizada para ejecutar de forma paralela un bucle mediante OpenMP:

```
#pragma omp parallel for schedule(kind [,chunk size])
```

donde `chunk` es la granularidad de las cargas de trabajo distribuidas entre los distintos hilos y `kind` el tipo de planificación (*scheduling*) mediante la cual serán distribuidas las tareas entre todos los subprocesos o hilos. Esta planificación tiene un importante impacto en el rendimiento final de las ejecuciones y debe ser escogida por el programador para cada fragmento paralelo identificado. En los bucles, esta cláusula determina cómo se asignan las distintas iteraciones entre los hilos de ejecución.

Existen además una serie de cláusulas adicionales que permiten controlar cómo se emplean las variables: `shared` (compartida) es utilizada para identificar aquellas que pueden ser compartidas entre los hilos de ejecución, `private` (privada) indica que cada hilo tendrá acceso exclusivo a la copia local de la variable y `reduction` (reducción), posibilita la especificación de cálculos recurrentes relacionados con operadores asociativos y conmutativos. En algunos casos, como por ejemplo en bucles con un gran número de instrucciones, el usuario puede necesitar bastante tiempo para distinguir las variables públicas de las privadas o escoger aquellas que conllevan operaciones de reducción.

Los cuatro tipos de planificación ofrecidos por OpenMP son los siguientes: `static` (estática), `dynamic` (dinámica), `guided` (guiada) y `runtime` (en tiempo de ejecución):

- Estática: es la planificación utilizada por defecto por la mayoría de compiladores cuando la cláusula es omitida, al ser la que menos *overhead* de comunicación produce cuando cada iteración emplea aproximadamente el mismo tiempo de ejecución. El total de iteraciones es organizado en diversos subconjuntos de fragmentos o trozos (`chunks`)[8] consecutivos de un tamaño especificado o lo más idénticos posibles[9], que son asignados estáticamente a los subprocesos de forma rotatoria, con un algoritmo de *Round-Robin* y siguiendo el orden de los hilos.

- Dinámica: permite manejar cargas de trabajo impredecibles o pobremente balanceadas (por ejemplo cuando estas cargas van disminuyendo durante el recorrido del bucle). Las iteraciones son asignadas desde la cola de trabajo a los hilos en el momento que estos solicitan otro fragmento, hasta que no existen más trozos. En concreto, el parámetro `chunk` define el número de iteraciones contiguas del bucle que ejecuta cada hilo. En caso de no especificarse su tamaño, el valor predeterminado es 1. Esta planificación puede aumentar significativamente el *overhead* cuando el tiempo de ejecución de cada uno de los fragmentos es demasiado pequeño. Sin embargo, tampoco es sencillo escoger el tamaño óptimo, dado que emplear trozos excesivamente grandes genera igualmente desequilibrios en las cargas de trabajo [10].

- Guiada: similar a la dinámica, en este caso, los tamaños de los fragmentos disminuyen con el tiempo, siguiendo la política de que para disminuir el *overhead* de comunicación es preferible utilizar trozos más grandes al comienzo de la ejecución. A medida que ésta avance, el sistema utilizará *chunks* más pequeños para llenar los huecos en la planificación, consiguiendo así un mayor aprovechamiento de los recursos. Para ello el tamaño de los fragmentos es siempre proporcional al número de iteraciones restantes dividido entre el número de hilos. En este caso, el `chunk_size` especificado en la directiva del bucle especifica el tamaño más pequeño que puede tener un fragmento, es decir, el número mínimo de iteraciones a asignar a cada hilo (por defecto, una).

- En tiempo de ejecución: realmente no se trata de un esquema de planificación. A diferencia del resto de tipos, en este caso la planificación es escogida durante la ejecución (y no en tiempo de compilación) a través de la variable de entorno `OMP_SCHEDULE`, mediante la cual el programador puede interactuar con el entorno de ejecución, para personalizar la planificación a aplicar.

Adicionalmente, también es posible gestionar la estrategia de forma manual, utilizando el número de identificación de cada hilo para asignar las diferentes cargas de trabajo.

Escoger la planificación más adecuada de antemano no resulta una tarea sencilla para el usuario, puesto que depende de una correcta comprensión de la ejecución paralela del bucle. La decisión se complica aún más en relación a la elección del valor para el tamaño de `chunk_size`. Una correcta selección depende de varios factores entre los que se incluyen el código incluido en el bucle, el tamaño del problema, el número de hilos utilizados y la propia carga del sistema. Además, los usuarios deben perseguir siempre un equilibrio entre

[8] Número que establece la cantidad de iteraciones consecutivas que se asignan a cada hilo.

[9] Como consecuencia del reparto, el último fragmento puede presentar menos iteraciones.

[10] Principalmente, se producen desequilibrios cuando varios hilos realizan tareas distintas, de manera que los más rápidos deben esperar hasta que los más lentos terminen y alcancen el mismo punto de ejecución.

el uso óptimo de la memoria y un balanceo de carga adecuado, analizando el rendimiento para averiguar qué método produce los mejores resultados. Una tarea que puede resultar realmente compleja para usuarios noveles en la programación paralela.

Como se describe en el apartado 2.2.4 sobre las medidas de rendimiento de los programas paralelos, en una ejecución paralela existe siempre un *overhead* asociado a la coordinación de las tareas: crear, iniciar y detener hilos, así como identificar qué trabajo debe realizar cada tarea. Además, se deben considerar los tiempos de espera en barreras, secciones críticas y cerrojos. En el caso de OpenMP los hilos tienen que ser generados o despertados del estado inactivo, hay que determinar el tamaño de los distintos paquetes de trabajo (en el caso de las planificaciones dinámica o guiada) y estos tienen que ser asignados a los hilos que vayan quedando libres durante la ejecución. Al finalizar las regiones paralelas, estos deben volver a unirse en el hilo maestro. Todo este *overhead* de comunicación implica que el *speedup* de una ejecución con OpenMP esté definido por la siguiente ecuación [63]:

$$S_{OMP}(N) = \frac{1}{s + \frac{1-s}{N} + \kappa N + \lambda} \tag{2.7}$$

siendo N el número de hilos, κN el *overhead* de comunicación y λ el *overhead* n-independiente (del número de hilos) y, a su vez,

$$\kappa = \frac{n}{B} \tag{2.8}$$

donde n es el tamaño del mensaje y B el ancho de banda. Por tanto, el *speedup* máximo alcanzable disminuye en función del número de hilos y del *overhead* generado por las ejecuciones paralelas.

De este modo, en ocasiones la paralelización de un bucle mediante directivas OpenMP implica un *overhead* excesivo que resulta inaceptable respecto a su versión secuencial. Como alternativa, existe la posibilidad de asignar explícitamente la carga de trabajo a cada hilo de ejecución. De esta forma se pueden obtener códigos altamente eficientes. Sin embargo, su implementación debe ser realizada manualmente por el programador, teniendo que asegurarse de evitar los posibles errores que produzcan, por ejemplo, problemas como interbloqueos[11], lo cual implica habitualmente una mayor complejidad.

En general, el rendimiento de un programa OpenMP está influenciado por los siguientes factores: accesos a memoria de cada hilo; instrucciones del código que son ejecutadas de forma secuencial; el *overhead* asociado a cada paralelización; el desequilibrio de carga entre los puntos de sincronización de los hilos; y las esperas para acceder a una región crítica o actualizar una variable implicada en una instrucción declarada como atómica [148].

[11] En un bloqueo muto o interbloqueo todos los hilos se esperan unos a otros, de forma que ninguno puede continúa su ejecución.

2.3.4. Aprendizaje automático aplicado a la paralelización OpenMP

La correcta paralelización de un código no garantiza las ejecuciones óptimas de los programas resultantes, sino que sus tareas paralelas deben ser asignadas de forma eficiente al correspondiente *hardware* para conseguir un rendimiento adecuado. Por ello, existen diversos enfoques que presentan como objetivo escoger la combinación de parámetros que permita alcanzar los mejores resultados.

Respecto a OpenMP, esta selección se focaliza principalmente en escoger la planificación más adecuada con objeto de obtener el mejor rendimiento en la ejecución paralela. Entre los distintos enfoques presentes en la literatura, varios se centran en la aplicación de técnicas de aprendizaje automático (*machine learning*), aunque aún no han sido ampliamente aplicadas en la programación paralela de bucles [149].

Estas técnicas posibilitan la utilización de modelos predictivos a partir de conjuntos de entrenamiento con datos específicos, que posteriormente son utilizados para realizar los pronósticos necesarios. Algunos de los algoritmos más empleados incluyen, entre otras, técnicas de regresión, árboles de decisión, inferencia bayesiana o redes neuronales artificiales. La posibilidad de aprendizaje de los distintos modelos facilita que estos puedan adaptarse a diferentes infraestructuras, cuyas características de computación y comunicación influyen notablemente en los resultados de las ejecuciones paralelas. En función de la forma de entrenamiento del modelo, el aprendizaje puede ser supervisado (los datos de entrenamiento están etiquetados previamente, de forma que se conoce cuál sería la previsión correcta para cada conjunto de datos) o no supervisado (se desconoce cuáles deben ser los resultados).

En [150] se ofrece una revisión sistemática que analiza y clasifica las propuestas existentes sobre técnicas de aprendizaje automático o metaheurísticas aplicadas a sistemas de computación paralela. Además, facilitan un navegador online que permite escrutar las publicaciones consideradas filtrando por método de optimización, actividad del ciclo de vida del *software*, arquitectura de computación, palabras clave y autores. En cualquier caso, se centran en todo tipo de propuestas y no solo en aquellas orientadas a escoger la planificación OpenMP más adecuada. Lo mismo sucede en [151], donde se presenta una revisión sistemática con menor profundidad de detalle que la anterior, que identifica aquellos enfoques que utilizan el aprendizaje automático y las metaheurísticas, clasificándolos según la infraestructura de cómputo, los métodos de optimización y la fecha de publicación. Asimismo, en [152] se realiza un estudio completo de las principales áreas de investigación y potenciales líneas futuras relacionadas con la aplicación de *machine learning* a la computación paralela, ofreciendo una detallada descripción de los principales conceptos sobre características, modelos, entrenamientos e implementación.

En general, los distintos enfoques se basan en analizar características estáticas y dinámicas del código para realizar estimaciones sobre su carga de trabajo [153–158]. Los datos correspondientes son asociados a los valores de rendimiento (principalmente en relación al *speedup* alcanzado), obtenidos mediante distintos esquemas o alternativas de ejecución, para posteriormente identificar patrones que permitan clasificar o predecir los resultados de futuras ejecuciones.

En [159] se propone un mecanismo adaptable basado en OpenMP que permite generar varias alternativas, con distinto número de hilos cada una, para un bucle determinado y seleccionar en tiempo de ejecución una versión adecuada. Para ello emplea el algoritmo de los *k* vecinos más próximos (*K-nearest neighbors*) [160]. Sin embargo, su aplicación se reduce a un experimento preliminar con dos únicos programas de ejemplo, sin haber implementado un aprendizaje automático completo.

En [161] se presenta un modelo para mejorar el rendimiento de programas basados en tareas OpenMP, prediciendo el esquema de planificación a partir de características generadas por el compilador y los tiempos de ejecución de ciertos fragmentos de código identificados para cada programa. Para ello clasifican cada instancia de entrenamiento añadiendo manualmente la estrategia correspondiente, que obtienen mediante la utilización del algoritmo no supervisado de *K*-medias (*K-means*). Posteriormente, para identificar la mejor planificación para cada nuevo código emplean las siguientes técnicas: clasificador bayesiano ingenuo (*Naive Bayes*) [162], redes neuronales artificiales de la clase perceptrones multicapas (ANN, *Multi-layer Perceptron Artificial Neural Network*) [163], máquinas de vectores de soporte multiclase (*multi-class* SVM, *Support Vector Machine*) [164] y árboles de decisión aleatorios (*random forest decision tree*) [165].

De forma similar, en [158] se ofrece un enfoque de modelado predictivo basado en la utilización de un SVM multiclase con un *kernel* que emplea funciones de base radial (*radial basis functions*), las cuales dependen de la distancia de un valor a un origen o centro determinado.

En [156] emplean un modelo de aprendizaje *off-line* que predice la planificación OpenMP más adecuada mediante análisis de rendimiento de bajo coste computacional. Para ello emplea máquinas SVM multiclase.

En [166] se utiliza un modelo de regresión logística para predecir el tamaño óptimo de los *chunks* para cada planificación mediante autoplanificación de fragmentos (*chunk self-scheduling*) [167].

De forma similar en [168] se propone una estrategia de decisión basada en árboles para acelerar aplicaciones relacionadas con el álgebra lineal. En concreto: suma de vectores, multiplicación de matriz por vector, y suma y multiplicación de matrices.

Recientemente, en [149] se ha presentado un algoritmo de planificación OpenMP adaptativo basado en emplear una estrategia de optimización bayesiana (*bayesian optimization*) [169], para ajustar los parámetros internos del algoritmo de factorización de Hummel (FSS, *Factoring Self-Scheduling*) [170], centrado en la planificación dinámica de bucles paralelos.

En relación a todas estas propuestas, es importante destacar su falta de generalización más allá de los distintos códigos utilizados en cada una, de forma que se desconoce su efectividad con otros programas, cargas de trabajo y especialmente respecto a otros dominios de aplicación [149]. En consecuencia, resulta complicado extrapolarlas a los casos planteados en el marco de esta investigación.

2.4. Técnicas para la escritura de códigos de programación eficientes

No existen demasiadas propuestas destinadas a escribir código eficiente en infraestructuras HPC, siendo habitual que muchos programadores no consideren la eficiencia de sus implementaciones a la hora de escribir sus códigos, independientemente del tipo de infraestructura en la que realicen sus ejecuciones.

En lo que a optimización de desarrollos C+ y C++ respecta, en [36] se presenta una propuesta centrada en el impacto de la optimización del compilador en la eficiencia de varios programas conocidos y usados en la literatura.

Fog proporciona en [171] una amplia guía de referencia sobre la optimización de C++ en plataformas Windows, Linux y Mac, considerando la familia x86 de microprocesadores de Intel, AMD y VIA, incluidas las versiones de 64 *bits*, y utilizando como medida de tiempo ciclos de reloj de CPU en lugar de segundos o microsegundos.

En [172] puede encontrarse un capítulo completo dedicado a la programación eficiente en C (sobre arquitecturas ARM (Advanced RISC Machine) en general, pero sin estimaciones de tiempo).

En [173] se presenta un documento técnico sobre el rendimiento en C++ independientemente del sistema operativo o el compilador empleado. Ofrece un modelo de *overheads* de espacio y tiempo implícitos en el uso de varias funciones del lenguaje y sus librerías. Además, presenta técnicas para emplear C++ en aplicaciones donde el rendimiento es importante y producir así código eficiente.

En [174] se proporciona un significativo número de ejecuciones de ejemplo que demuestran cómo aplicar principios de optimización del rendimiento para mejorar código C++, ofreciendo también mediciones de tiempo de los resultados en un procesador i7 (Intel Corporation, Santa Clara, UT, EE.UU.) con Visual Studio 2010 Compiler (Microsoft Corporation, Redmond, WA, EE.UU.).

En [175] se introduce un sistema de optimización para evitar código muerto (*dead code*)[12] y subexpresiones comunes (CSE, *Common Subexpression Elimination*)[13], mientras que en [176] se describen algunos métodos para la eliminación de código muerto y la aplicación de técnicas de *inlining*[14] en lenguaje C/C++.

En [177] se presenta un conjunto completo de técnicas de mejora de código basadas en el compilador y centradas principalmente en código C, que describe técnicas analíticas y

[12] Instrucciones de memoria que nunca pueden efectuarse en tiempo de ejecución o cuyos resultados no son utilizados en ningún otro cómputo.

[13] La eliminación de subexpresiones comunes (CSE) consiste en reemplazar instancias de expresiones idénticas (por ejemplo, aquellas que evalúan el mismo dato) por una sola variable que contenga el valor adecuado.

[14] Optimización que reemplaza la llamada a una función con el propio contenido de dicha función.

problemas específicos de flujo de datos y evalúa transformaciones para mejorar los tiempos de ejecución de programas en máquinas monoprocesador.

En [178–180] pueden encontrarse reglas generales y técnicas utilizadas para optimizar programas en lenguaje C.

Algunas propuestas recientes han sido centradas en técnicas de refactorización[15] y reestructuración de códigos [181–183] sin cambiar el comportamiento externo, pero solo cubren un número limitado de técnicas y no contienen suficientes detalles para ayudar a los desarrolladores a escribir códigos más eficientes.

En cualquier caso, existe una ausencia de enfoques que analicen las ventajas de escribir manualmente código eficiente para infraestructuras HPC. Por ello, la investigación realizada en la presente tesis doctoral proporciona un conjunto de tests que han sido desarrollados exclusivamente para la medición y análisis de las veintiséis técnicas diferentes propuestas para reducir los tiempos de ejecución, con el objetivo de demostrar la importancia de aplicar ciertas estrategias eficientes para desarrollar códigos C/C++ para este tipo de sistemas. Estas técnicas han sido escogidas como las más representativas de las existentes en la literatura, por lo que pueden ser aplicadas sobre una amplia gama de rutinas y tareas de programación, mejorando de forma destacable los tiempos de ejecución de las aplicaciones correspondientes. Además, son fáciles de utilizar tanto para programadores avanzados, como para principiantes.

[15] La refactorización tiene como objetivo mejorar el diseño, la estructura y la implementación del código preservando su funcionalidad.

Capítulo 3

Transcompilador para la paralelización automática de códigos

En este capítulo se presenta el transcompilador propuesto para la paralelización automática de códigos de programación secuenciales. En primer lugar, en la sección 3.1 se describen tanto el alcance como los objetivos específicos de la aportación realizada. Posteriormente, en el apartado 3.2 se expone la implementación desarrollada y se detallan en profundidad los distintos módulos que conforman el transcompilador. Además, se ofrece un ejemplo de paralelización de un código fuente secuencial basado en la multiplicación de una matriz por un vector.

3.1. Alcance y objetivos del transcompilador

Científicos e investigadores de todo el mundo se enfrentan diariamente a desafíos críticos que requieren del uso de la computación de alto rendimiento, para obtener ciertos resultados en el menor tiempo posible, o alcanzar niveles de rendimiento computacional difícilmente alcanzables en otro tipo de sistemas [62]. En estos casos resulta clave la utilización de la computación paralela, que permite realizar una cantidad realmente significativa de cálculos de forma simultánea y extremadamente rápida, con la consecuente mejora e impacto positivo en sus proyectos [184]. Desarrollar programas que aprovechen los beneficios del HPC a menudo supone serias dificultades, especialmente para investigadores que no son expertos en programación paralela o que requieren adaptar sus códigos secuenciales, precisando así unos conocimientos altamente específicos. Consecuentemente, el transcompilador desarrollado presenta como objetivo mejorar el rendimiento y la eficiencia en los centros de supercomputación, considerando la gestión de recursos, para que incluso usuarios principiantes puedan hacer un uso adecuado de los mismos, minimizando los tiempos de ejecución asociados.

El transcompilador[1] posibilita la paralelización automática de códigos secuenciales centrando su objetivo en los bucles. La selección de estas estructuras de control como objetivos principales de la paralelización, ha sido motivada por el hecho de tratarse del enfoque más empleado para distribuir tareas computacionales entre distintos hilos de procesamiento [63]. Además, es habitual que gran parte de programas y aplicaciones científicas consuman en los bucles la mayor parte de sus tiempos de ejecución. Los lenguajes de programación C/C++ han sido escogidos por ser ampliamente utilizados por los usuarios de la computación de alto rendimiento [147], aunque se prevé que otros lenguajes adicionales sean considerados en futuras mejoras del trabajo desarrollado. Cabe destacar asimismo, que los códigos paralelos generados por el transcompilador mantienen intacto todo el código secuencial original, debido a que la paralelización automática es definida mediante la introducción de directivas de compilación. De este modo, los programas resultantes pueden ser ejecutados de forma tanto secuencial como paralela, ayudando así a potenciar el aprendizaje de aquellos programadores con menos experiencia en la implementación de este tipo de algoritmos.

El uso de lenguajes interpretados, como Java y Python, ha sido reservado para futuros trabajos porque requieren ser traducidos durante el tiempo de ejecución. En cambio, los lenguajes compilados incurren una única vez en esta sobrecarga al compilar la fuente y por lo tanto, dependiendo de la aplicación, con ellos es posible conseguir mejores tiempos de ejecución más reducidos [35]. En este sentido, otros como Fortran también hubieran supuesto una elección adecuada. En cualquier caso, los resultados obtenidos en la investigación pueden ser extrapolados a otros lenguajes de programación, aunque esto requiere estudios y análisis adicionales que aún no han sido desarrollados y que serán abordados en el futuro.

3.2. Implementación

La Figura 3.1 muestra un esquema general del flujo de trabajo del transcompilador. En ella se distinguen cuatro partes principales: una primera dedicada a los análisis léxico, sintáctico y semántico del código secuencial de entrada (apartado 3.2.1); la segunda y la tercera en las que se analiza la información obtenida y se genera el código paralelo (apartados 3.2.2 y 3.2.3); y finalmente, una cuarta sección dedicada a la optimización del mismo (3.2.4), asistida por un sistema de apoyo a la toma de decisiones (3.2.4.1).

En primer lugar, se realiza un análisis lexicográfico (*scanner*) que descompone en componentes básicos (*tokens*[2] o símbolos) el código C del programa secuencial a paralelizar, los cuales son procesados posteriormente por el analizador sintáctico (*parser*),

[1] Un compilador fuente a fuente (*source-to-source compiler*) o transcompilador (*transcompiler*), es un tipo de traductor que transforma un código fuente en otro equivalente, empleando idéntico lenguaje de programación o uno similar situado en el mismo nivel de abstracción (a diferencia de un compilador tradicional, que traduce un lenguaje de nivel superior a otro de nivel inferior). En la literatura predomina la denominación *source-to-source compiler*, pero en esta investigación se ha priorizado el uso de la palabra "transcompilador" en su designación, por considerarla más adecuada para su alusión en español a lo largo del presente documento.

[2] Palabras reservadas de un lenguaje que representan una unidad de información específica, como por ejemplo, <FOR : "for">.

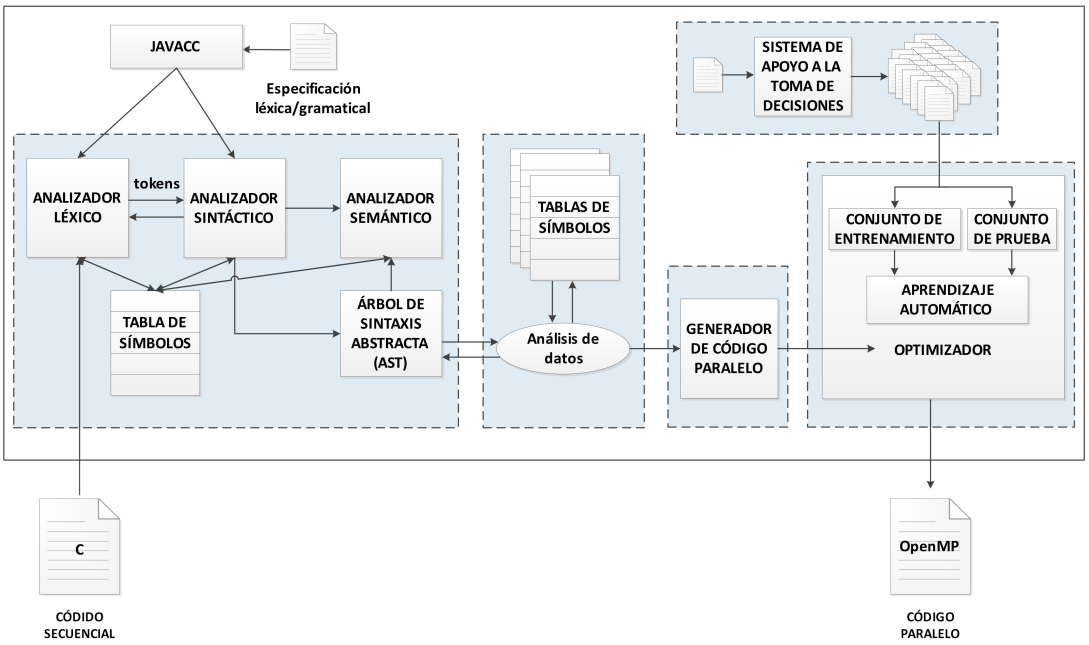

Figura 3.1: Flujo de trabajo del transcompilador.

que a su vez comprueba que la estructura sea correcta según la gramática del lenguaje y genera un árbol AST con su representación.

Tras este paso se realiza un análisis del AST mediante un patrón de comportamiento de diseño *software* y varias tablas de símbolos, obteniendo todos los datos necesarios.

A continuación, el módulo generador transforma el código secuencial en código paralelo OpenMP, utilizando para ello la información obtenida en el paso anterior. Finalmente, el optimizador escoge la mejor planificación OpenMP para el código resultante, gracias a un módulo de aprendizaje supervisado entrenado con la ayuda del DSS.

3.2.1. Lectura del código fuente

El *parser*, a partir de los *tokens* que recibe del analizador léxico, debe comprobar que la estructura del código fuente se ajusta a la especificación sintáctica del lenguaje secuencial de entrada. Inicialmente se consideró el uso de Clang [185] para su creación, sin embargo, fue descartado por estar centrado únicamente en la familia de lenguajes de C[3], puesto que en futuros trabajos se prevé ampliar el transcompilador para que admita otros lenguajes de programación adicionales. Finalmente, el *parser* ha sido creado utilizando el metacompilador JavaCC (*Java Compiler Compiler*) [186], que integra funciones de análisis léxico y sintáctico y genera, a partir de una especificación gramatical dada y de forma automática, el código que permite reconocer coincidencias con dicha gramática. JavaCC proporciona además otras capacidades estándares relacionadas con la generación

[3]Clang proporciona un entorno de herramientas y un *front-end* para los siguientes lenguajes de programación: C, C++, Objective C/C++, OpenCL, CUDA y RenderScript.

de *parsers* como, por ejemplo, la construcción de árboles (mediante la herramienta JJTree incluida en JavaCC), la inclusión de acciones o las capacidades de depuración. A continuación se resumen sus principales características [186]:

- Genera un analizador sintático descendente (*top-down parser*, que simula la expansión del árbol sintáctico comenzando desde la raíz o axioma y descendiendo hasta las hojas) y recursivo (debido a la naturaleza recursiva de la propia gramática), a diferencia de otros analizadores similares a YACC [187] que emplean análisis ascendente[4]. Aunque esto haga que la recursividad por la izquierda[5] sea inviable, permite a su vez la utilización de gramáticas más generales, facilita las tareas de depuración, ofrece la capacidad de analizar cualquier símbolo no terminal[6] y posibilita poder pasar valores (atributos) hacia arriba o hacia abajo del árbol durante el análisis.

- De forma predeterminada JavaCC genera un analizador LL(1)[7]. Sin embargo, pueden existir partes de la gramática que no sean LL(1). Presenta capacidades de anticipación sintáctica y semántica para resolver localmente las ambigüedades desplazamiento-desplazamiento (*shift-shift ambiguities*) en estos puntos de conflicto. Es decir, permite que el analizador sea LL(k) solo en dichos puntos, pero sigue siendo LL(1) en el resto de producciones para obtener un mejor rendimiento. Además, cabe recordar que los analizadores descendentes como JavaCC, son ajenos a los problemas desplazamiento-reducción (*shift-reduce*) y reducción-reducción (*reduce-reduce*).

- Los *parsers* creados con JavaCC no presentan dependencias en tiempo de ejecución, al ser generados exclusivamente con lenguaje Java.

- Permite emplear especificaciones BNF (*Backus-Naur Form*[8], notación de Backus-Naur) extendidas, posibilitando la utilización de símbolos propios de expresiones regulares, como "`(A)`*" y "`(A)`$^+$", dentro del léxico y las especificaciones gramaticales, mitigando en cierto modo la necesidad de recursividad por la izquierda y facilitando su lectura, como por ejemplo al utilizar "`A ::= y(x)`*" en lugar de "`A ::= Ax|y`".

- Las especificaciones léxicas y sintácticas son registradas en el mismo archivo, mejorando la legibilidad y el mantenimiento de las gramáticas, ya que es posible insertar expresiones regulares (léxicas) en las especificaciones gramaticales.

[4] En los *parsers* ascendentes (*down-top parsers*) el análisis se realiza de abajo hacia arriba, comenzando desde los nodos de las hojas del árbol y trabajando en dirección al axioma. Se parte de una sentencia y posteriormente se aplican las reglas de producción de manera inversa para llegar al símbolo inicial. Es decir, parte del objetivo y trata de reconstruir las producciones hasta alcanzar el axioma.

[5] Producida cuando un símbolo no terminal contiene una referencia recursiva a sí mismo, sin estar precedido por nada que utilice *tokens*.

[6] Símbolos más abstractos, que pueden ser reemplazados y no se incluyen directamente en las cadenas del lenguaje, pero que representan posibles partes de las mismas. Pueden ser denominados variables sintácticas.

[7] Las gramáticas LL(1) permiten el análisis determinista descendente sin evaluar más *tokens* que el actual en cada momento. El nombre se deriva de sus características: la primera "L" significa "izquierda" e indica que la cadena de entrada se analiza de izquierda a derecha; la segunda "L" hace referencia a *leftmost*, más a la izquierda, y señala que el analizador determina la derivación más a la izquierda; el número 1 expresa que en cada paso del análisis, el parser sólo necesita ver el siguiente *token* para determinar sus decisiones [188].

[8] Metalenguaje empleado para expresar gramáticas libres de contexto.

- Su analizador léxico posibilita el uso de caracteres Unicode, facilitando la descripción de elementos del lenguaje con caracteres no ASCII como, por ejemplo, los que pueden ser utilizados en los identificadores de Java.
- Aunque ofrece funciones y estados similares a Lex [189], incorpora conceptos adicionales que permiten la utilización de especificaciones más limpias y mejoran la gestión de mensajes de error y advertencia. JavaCC considera: *tokens* normales (`TOKEN` crea un *token* a partir de un lexema[9] o cadena coincidente y lo envía al *parser*); espaciadores (`SKIP` indica los *tokens* que no serán enviados al analizador sintáctico como, por ejemplo, espacios, tabuladores o retornos de carro); especiales (`SPECIAL TOKEN`, semejante a `SKIP`, pero en este caso los lexemas son conservados, de forma que pueden ser utilizados. Esto permite, por ejemplo, mantener los comentarios del código fuente original); y de seguimiento (`MORE` continúa hasta el siguiente lexema y lo concatena al actual) [190].
- Ofrece información de diagnóstico completa en su gestión de errores, permitiendo localizar fácilmente la ubicación de errores sintácticos.
- Permite acciones de depuración, tanto en el *parser* como en el analizador lexicográfico, con las opciones `DEBUG_PARSER` (obtiene información de depuración del *parser* en tiempo de ejecución), `DEBUG_LOOKAHEAD` (indica el número de *tokens* considerados por el *parser* antes de tomar una decisión, siendo LL(1) el valor por defecto) y `DEBUG_TOKEN_MANAGER` (permite analizar el procesamiento de los *tokens*) [190].
- Es posible personalizar tanto el propio comportamiento de JavaCC como el relativo a los analizadores sintácticos generados.

En primer lugar, se ha utilizado la notación de Backus-Naur (EBNF, *Extended Backus-Naur Form*) [191] para especificar la gramática C. Posteriormente, y tras realizar una ampliación de la especificación que permite la extracción de la información adicional necesaria para paralelizar cada código, la notación EBNF ha sido convertida al correspondiente archivo de especificación de JavaCC, en base al cual es generado el código de ejecución del *parser*, posibilitando así el procesamiento del lenguaje a partir de su gramática. Como resultado, el *parser* ha sido integrado en el contexto del transcompilador.

El código 3.1 muestra como ejemplo la especificación EBNF para una estructura de control de bucle `for`. Los paréntesis alrededor de "`(type)? expression`" conllevan que su contenido sea tratado como una unidad. Las interrogaciones "`?`" expresan opcionalidad, mientras que el signo "`+`" denota una o más ocurrencias en lugar de cero o más.

Puesto que JavaCC realiza un descenso recursivo, no permite analizar grámaticas con recursividad por la izquierda para prevenir que las subrutinas generadas se llamen a sí mismas de forma reiterada e infinita. De hecho, muestra mensajes de error cuando existen producciones recursivas por la izquierda. Por ello, éstas han sido transformadas antes de escribir las reglas gramaticales. El enfoque general ha consistido en reemplazar las reglas de la forma "`A → a | Ax`", por otras reglas con forma "`A → (x)*`", como muestra el ejemplo del código 3.2.

[9] Secuencia de caracteres que coincide con una expresión regular. Por ejemplo, *'F' 'O' 'R'* encaja con el lexema `FOR`.

Código 3.1: Especificación EBNF para la estructura de un bucle "for".

```
1 forStatement := ( ( <FOR> ) <LEFT_PARENTHESIS>
2                 ( ( type )? expression )? <SEMICOLON>
3                   ( expression )? <SEMICOLON>
4                   ( expression )? <RIGHT_PARENTHESIS>
5                 statement )
```

Código 3.2: Ejemplo de transformación de recursividad por la izquierda.

```
1 identifierList := identifier ( <COMMA> identifier )*
```

El archivo de especificación gramatical de JavaCC presenta tres componentes principales: las definiciones de los *tokens* terminales, las definiciones de las reglas de producción que establecen cómo combinar dichos *tokens* para formar declaraciones aceptadas por el lenguaje y finalmente, el código Java requerido para realizar acciones especiales cuando se procesan determinas producciones.

De este modo, durante esta primera fase el transcompilador lee el código fuente secuencial de entrada como un archivo de caracteres, los cuales son agrupados en *tokens*, que representan los distintos patrones reconocidos. Por ello, la salida de este primer análisis está formada por secuencias de componentes léxicos que posteriormente son procesadas por el analizador sintáctico y que representan palabras reservadas, identificadores, operadores, símbolos especiales, constantes numéricas y de carácter. Así, estos componentes van siendo enviados al *parser* bajo demanda de éste, asegurando que la construcción de los *tokens* es correcta de acuerdo con la especificación sintáctica.

Además, con objeto de poder analizar adecuadamente el código secuencial de entrada y dada la complejidad inherente al lenguaje C, es necesario convertirlo a una interpretación intermedia más sencilla. Para ello, tras realizar los correspondientes análisis léxico y sintáctico, se emplea un árbol de sintaxis abstracta que representa el código secuencial. Esto permite al analizador comprobar el correcto uso de los elementos de entrada durante el análisis semántico por medio del AST, que además proporciona la base para la posterior generación de código. La separación de ambos análisis, léxico y sintáctico, facilita a su vez la realización de posibles cambios posteriores en el transcompilador.

3.2.2. Análisis de datos

Aunque la estructura del AST permitiría analizar directamente el código secuencial de entrada, se ha decidido facilitar el recorrido del árbol mediante la implementación de un patrón de diseño de comportamiento, con el propósito de poder definir nuevas operaciones sobre su estructura sin modificar las clases de los objetos sobre los que opera, es decir, sin que sea necesario recompilar estas clases. De este modo, la utilización del patrón evita la necesidad de embeber dentro de las clases del árbol, la funcionalidad de extracción de la información que posibilita paralelizar posteriormente y de forma automática el código.

Asimismo, el núcleo del patrón, es decir su interfaz, posibilita definir una operación para cada tipo de nodo del árbol. Así, el extractor de información está por tanto, compuesto por varias clases que atraviesan el AST para extraer y analizar los datos requeridos para la paralelización.

La información adquirida durante el recorrido del AST es registrada en varias tablas de símbolos, cada una con una estructura de datos *hash*, organizadas a su vez en una estructura general de tipo pila. En ellas, cada identificador obtenido del código de entrada (variables, funciones, tipos, etc.) es asociado con su correspondiente información relacionada, como por ejemplo, nombre, valor contenido, tipo de identificador, ámbito o ubicación exacta en el código fuente, entre otros datos. Esto permite separar o distinguir un identificador para saber si se trata de una función o una variable, conocer su memoria asignada o la dirección en la cual es almacenado.

3.2.3. Generador de código paralelo

En el siguiente módulo, el generador de código procesa la información extraída del AST junto con la registrada en las tablas de símbolos, con el objetivo de transformar el código secuencial de entrada en código paralelo. Para ello se implementan directivas OpenMP que son introducidas en el código original. El generador evalúa los bucles secuenciales existentes y define la construcción de las directivas y las cláusulas asociadas necesarias, determinando asimismo qué variables del bucle deben ser compartidas y cuáles pueden ser privadas.

El generador tiene siempre en cuenta una serie de consideraciones para realizar la paralelización. En primer lugar, la ejecución en paralelo de un programa secuencial implica siempre un *overhead* o tiempo adicional para la coordinación de las tareas (como se describe en los apartados 2.3.3 y 2.2.4). Por ello, con el objetivo de minimizar estos tiempos, en los bucles anidados el generador de código persigue siempre la paralelización del bucle más externo (evitando en la medida de lo posible paralelizar exclusivamente el contenido de los bucles internos). De este modo se evitan situaciones en las que, por ejemplo, para la ejecución de una región paralela en un bucle doblemente anidado se incurriría $n \times m$ veces en este *overhead* (siendo n y m el número de iteraciones de cada bucle).

Adicionalmente, respecto a la determinación de qué datos deben ser compartidos entre hilos de ejecución y cuáles deben ser locales a un único subproceso, aunque OpenMP proporciona una configuración automática, es recomendable especificar cada atributo de intercambio de datos de forma explícita [148], en lugar de confiar en la configuración por defecto. Así se reduce notablemente la posibilidad de provocar errores y comportamientos no deseados. Por ello, el generador de código prioriza la utilización de variables privadas siempre que sea posible, dado que también se recomienda minimizar el uso de variables compartidas para alcanzar un mejor rendimiento [148].

En el ejemplo de código 3.3 [148] puede observarse cómo al no declararse explícitamente una variable como privada (en este caso, x), y por tanto ser compartida por defecto, se producen condiciones de carrera. La variable x puede ser escrita

Código 3.3: Ejemplo de variable compartida por defecto.

```
1  void compute(int n) {
2    int i;
3    double h, x, sum;
4    h = 1.0/(double) n;
5    sum = 0.0;
6    #pragma omp for reduction(+:sum) shared(h)
7    for (i=1; i <= n; i++) {
8       x = h*((double)i-0.5);
9       sum += (1.0/(1.0+x*x));
10   }
11   pi = h * sum;
12 }
```

simultáneamente por varios hilos, de forma que cada cambio realizado en uno afecta al valor de x en el resto. En caso de ser declarada privada, cada hilo tendría su propia copia y podría presentar simultáneamente distintos valores en cada uno. Cabe destacar sin embargo, que las reglas implícitas sobre los atributos de intercambio de datos de OpenMP, establecen que las variables de iteración de un bucle `for` (en este ejemplo, i) sean siempre privadas, aunque en el caso de bucles anidados esto se aplica únicamente al nivel más externo. Además, el generador siempre considera como privadas aquellas variables que son declaradas localmente dentro de una región paralela.

El generador determina qué bucles pueden ser realmente paralelizados, evaluando para ello si existen condiciones de carrera y determinando si cada iteración del bucle puede ser ejecutada independientemente del resto. Así, el ejemplo de código 3.4 no sería paralelizado, al contener una dependencia entre instrucciones ejecutadas en diferentes iteraciones del bucle. En general, no es sencillo detectar si este hecho se produce. Cuanto más complejo es el código contenido en el bucle, más complicado resulta para un usuario garantizar sin la ayuda del transcompilador, que no se introducen este tipo de errores. A medida que aumenta esta complejidad, crecen las posibilidades de que el programador paralelice su código manualmente sin ser consciente de que múltiples iteraciones hacen referencia a un mismo elemento, especialmente si se trata de bucles con varios niveles de anidación.

Las condiciones de carrera originan un comportamiento no determinista, siendo posible que el error surgido por una paralelización inadecuada sea enmascarado durante las pruebas del código. Esto puede deberse, entre otros motivos, a que por ejemplo el número

Código 3.4: Ejemplo de bucle con condiciones de carrera.

```
1  for (i=0; i<n; i++) {
2     a[i] = a[i] + c;          (I1)
3     b[i] = a[i-1] * b[i];     (I2)
4  }
```

de hilos de ejecución no permita que el problema se genere y, sin embargo, al escalar el programa se originen los inconvenientes.

En este ejemplo, aunque no existen dependencias entre las instrucciones I_1 e I_2 en una única iteración del bucle, sí que se producen entre dos iteraciones sucesivas, puesto que cuando $i = k$, I_2 lee el valor de $a[k-1]$, escrito por I_1 en la iteración $k-1$. En este caso, si se paralelizase el bucle insertando una directiva `parallel for`, podrían generarse resultados erróneos. Por ello, para detectar si existen dependencias en el nucleo del bucle, el generador determina cuándo en una iteración una instrucción actúa sobre una variable que es leída o escrita en el resto de iteraciones.

En el Algoritmo 1 se muestra el pseudocódigo utilizado para identificar este tipo de dependencias en los bucles, siendo `leftArrays` y `rightArrays` dos listas que se generan mientras se atraviesa el AST, cada vez que se detecta un operador de asignación. Ambas contienen información sobre los vectores involucrados en cada instrucción (nombre, tipo, expresión e índice, entre otros datos) y son almacenadas en la correspondiente tabla de símbolos del bucle o en la referente al más externo en el caso de tratarse de bucles anidados.

El generador también proporciona, de forma automática y en caso necesario, la cláusula de reducción ofrecida por OpenMP para especificar ciertos cálculos recurrentes. Estos involucran operadores matemáticamente asociativos y conmutativos, de modo que la claúsula permite que puedan ejecutarse en paralelo sin modificar el código. De esta forma, el generador identifica las operaciones y las variables que contendrán los resultados y emplea la sintaxis `reducción (operador: lista)` para conseguir la paralelización automática.

El pseudocódigo del algoritmo general desarrollado para decidir cómo construir las directivas *pragma* de cada bucle, es mostrado en el Algoritmo 2. Las variables son identificadas y clasificadas (en compartidas, privadas o de reducción) mientras se atraviesa el AST, cada vez que se detecta un identificador dentro de un bucle `for`. De esta manera, como se observa en el pseudocódigo, el procedimiento `generatePragmaDirective`, tras comprobar que realmente existe el bucle a paralelizar (líneas 2 y 3 del algoritmo), busca en primer lugar las posibles dependencias, tanto en el cuerpo del propio bucle (líneas 4 y 5) como en los bucles internos anidados que existan (líneas 6 a 8). A continuación, para generar la directiva de paralelización correspondiente al bucle más externo, analiza sus tres listas de variables (`privateVars`, `sharedVars` y `reductionVars`), teniendo en

Algoritmo 1: Análisis de dependencias en bucles entre distintas iteraciones.

```
1 Función lCDependence(leftArraysList, rightArraysList):
2     para i ← 0 hasta leftArraysList.size hacer
3         leftArray ← leftArraysList.get(i);
4         para j ← 0 hasta rightArraysList.size hacer
5             rightArray ← rightArraysList.get(j);
6             si leftArray.name es igual a rightArray.name entonces
7                 si rightArray.expression contiene leftArray.index entonces
8                     si leftArray.iteration no es igual a rightArray.iteration entonces
9                         devolver True
```

Algoritmo 2: Generación de directivas pragma OpenMP.

```
1  Función generatePragmaDirective(ForLooop forLoop):
2      si existOuterLoop entonces
3          devolver null
4      si LCDependence entonces
5          lanzar Exception: LoopCarried Dependence;
6      innerLoops ← forLoop.getInnerLoops;
7      si innerLoops.LCDependence entonces
8          lanzar Exception: LoopCarried Dependence;
9      mientras innerLoops.hasMoreElements hacer
10         loop ← innerLoops.nextElement;
11         privateVars ← loop.getPrivateVars;
12         sharedVars ← loop.getSharedVars;
13         reductionVars ← loop.getReductionVars;
14     para i=0 hasta reductionVars.size hacer
15         variable ← reductionVars.get(i);
16         si variable en sharedVariables entonces
17             sharedVars.delete(variable);
18         si variable existe en privateVars entonces
19             privateVars.delete(variable);
20     para i=0 hasta privateVars.size hacer
21         variable ← privateVars.get(i);
22         si variable existe en sharedVariables entonces
23             sharedVars.delete(variable);
24     devolver CreatePragma(private,shared,reduction);
```

cuenta que una variable de reducción no puede ser compartida o privada. Lo mismo ocurre con las variables privadas, que no se pueden compartir. Esto se debe a la forma estática en la que se clasifican las variables mediante la función `identifierAnalysis` (ver Algoritmo 3), la cual es llamada cada vez que se encuentra un nodo de tipo identificador durante el recorrido del AST.

Adicionalmente, este último procedimiento también distingue si el identificador pertenece a una función o una directiva de preprocesador de tipo `#DEFINE`, entre otras posibilidades, pero esto ha sido omitido en el pseudocódigo por razones de simplicidad.

Algoritmo 3: AST Identifier Node analysis.

```
1  Función identifierAnalysis(ASTNode identifier, ForLoop loop):
2      readOnly ← true;
3      reduction ← false;
4      si identifier.IsBeingUpdated() entonces
5          readOnly ← false;
6      si insideReduction(identifier) entonces
7          reduction ← true;
8          operator ← assignmentExpression;
9      variable ← NewVariable(identifier,readOnly,reduction);
10     variable.setReductionOperator(operator);
11     loop.add(variable);
```

3.2.3.1. Ejemplo de paralelización: multiplicación de matriz por vector

Con el objetivo de describir de forma detallada la paralelización automática realizada por el transcompilador, se utiliza como ejemplo básico la multiplicación de una matriz por un vector, una operación muy común empleada frecuentemente en el ámbito de la computación científica [192]. El programa secuencial calcula el siguiente producto:

$$Ab = c \tag{3.1}$$

siendo $A \in \mathbb{R}^{n \times m}$ una matriz $n \times m$, y $b \in \mathbb{R}^m$ un vector de tamaño m. De este modo, el producto de la matriz por el vector puede definirse como:

$$c_i = \sum_{j=1}^{m} a_{ij} b_j, \qquad i = 1, ..., n \tag{3.2}$$

considerando:

$$c = (c_1, ..., c_n) \in \mathbb{R}^n \tag{3.3}$$

$$A = (a_{ij})_{i=1,...,n; j=1,...,m} \tag{3.4}$$

$$b = (b_1, ..., b_m) \tag{3.5}$$

Por tanto, la operación puede ser implementada secuencialmente como se muestra en el Código 3.5 [148]. El bucle interno calcula el producto escalar de la fila i de la matriz a con el vector b y el resultado es almacenado en el elemento i del vector c. Este producto es realizado para todas las filas de la matriz, recorridas mediante el bucle externo. Aunque el vector a debe ser declarado y usado como una matriz lineal por razones de rendimiento, se utiliza una bidimensional con el objetivo de facilitar la comprensión del problema.

Para paralelizar el código, el transcompilador evalúa el código de entrada y realiza la paralelización sobre el bucle que itera sobre el índice i, al detectar que cada producto escalar calcula un elemento distinto del resultado (c) de forma independiente y el orden en que se calculan los elementos de $c[i]$ no afecta a la solución. En cualquier caso, siempre se evalúan todos los bucles existentes en el código, descartando además la posibilidad de que existan condiciones de carrera.

En el Código 3.6 puede verse el resultado de la paralelización automática realizada por el transcompilador, que inserta automáticamente una directiva *pragma OpenMP* (conjuntamente con un "`#include <omp.h>`" en la cabecera) y añade adecuadamente las cláusulas necesarias, de forma que durante la ejecución se crea una región paralela en

Código 3.5: Multiplicación de una matriz por un vector. Código secuencial.

```
1 for (i=0; i<n; i++){
2   c[i] = 0.0;
3   for (j=0; j<m; j++){
4     c[i] += a[i][j]*b[j];
5   }
6 }
```

Código 3.6: Multiplicación de una matriz por un vector. Código paralelo.

```
#pragma omp parallel for num_threads(48) schedule(static) default(none)
  private(i,j)
  shared(m,a,n,b,c)

  for (i=0; i<n; i++){
    c[i] = 0.0;
    for (j=0; j<m; j++){
      c[i] += a[i][j]*b[j];
    }
  }
```

la que se especifica que las iteraciones del bucle deben ser distribuidas entre los hilos de ejecución. Asimismo, se indica que cada uno de los hilos debe disponer de acceso exclusivo a una copia local de las variables de iteración i y j, mientras que las variables a, b, c, m y n pueden ser compartidas entre ellos, estableciendo por defecto la planificación estática.

3.2.4. Optimizador

El módulo de optimización ha sido desarrollado principalmente con objeto de ayudar a predecir la planificación más adecuada para ejecutar cada código paralelo generado por el transcompilador. Esto puede lograrse gracias a la aplicación de algoritmos de aprendizaje supervisado (entrenados asimismo por códigos paralelos previamente ejecutados en la infraestructura de cómputo), que pronostican cuál es la mejor estrategia de programación para cada bucle. Estos algoritmos podrán además estar orientados a predecir, no solo la planificación OpenMP más apropiada, sino también el número de núcleos necesarios para realizar cada ejecución paralela alcanzando una eficiencia correcta, con objeto de no desperdiciar recursos de cómputo.

En los siguientes apartados se describen las distintas partes que conforman el módulo de optimización, considerando especialmente el DSS y su módulo de aprendizaje automático.

3.2.4.1. Sistema de apoyo a la toma de decisiones (DSS)

Independientemente del aprendizaje supervisado que se emplee en el transcompilador, el sistema necesita ser entrenado previamente con conjuntos de datos, consistentes en códigos paralelos que ya contienen la planificación más adecuada para su ejecución. Con objeto de facilitar la creación de estos conjuntos de entrenamiento, se propone un sistema DSS para ayudar en la toma de decisiones (ver Figura 3.2), que permite de manera automática: modificar, compilar y enviar los trabajos de cómputo a las colas de ejecución HPC, así como extraer, ordenar y analizar los resultados de rendimiento obtenidos para cada código. Esto facilita significativamente seleccionar aquellos que presentan unos parámetros óptimos en función de una carga de trabajo o una eficiencia determinadas.

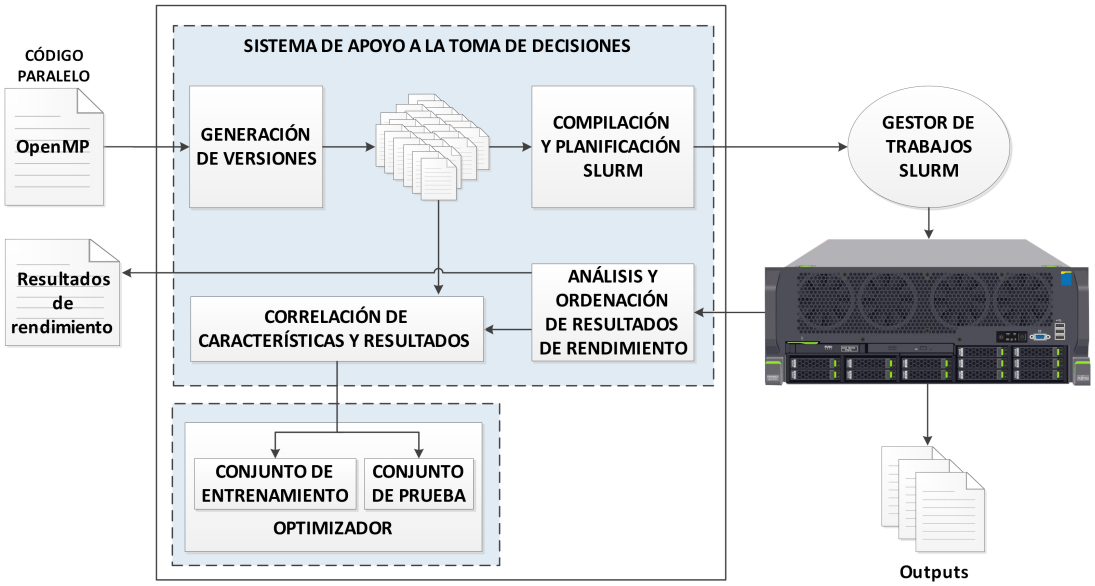

Figura 3.2: Flujo de trabajo del DSS.

En cuanto al envío automático de trabajos, el DSS ha sido implementado para trabajar con Slurm (ver apartado 2.1.4) como gestor de colas para acceder a los recursos o nodos de cómputo.

En primer lugar, el DSS desarrollado permite procesar un programa OpenMP y a partir de él producir y compilar de forma automática programas similares, pero con variaciones en las siguientes características (ver Figura 3.3):

- Distintas cargas de trabajo basadas en variable específicas (en este caso, seleccionadas manualmente por el usuario). Por ejemplo, la dimensión de la matriz en un programa de multiplicación de matrices cuadradas.
- Variaciones en el número de núcleos de procesamiento utilizados por la infraestructura para ejecutar el código, considerando todas las opciones posibles, es decir, ejecución con 1 núcleo o 2 núcleos o 3... y así sucesivamente, hasta alcanzar el número máximo de núcleos disponibles.
- Utilización de las tres estrategias de planificación de OpenMP: estática, dinámica y guiada.

A continuación el DSS compila, también de forma automática, estos programas similares produciendo todos los ejecutables binarios correspondientes, así como los archivos por lotes necesarios para enviar cada uno de los trabajos a la infraestructura de cómputo. Esto es esencial dado que, por ejemplo, considerando exclusivamente las estrategias de planificación y el número de núcleos de ejecución para un único programa y una infraestructura de destino con solo 20 núcleos, un usuario tendría que configurar, compilar y preparar de forma manual la ejecución de 60 versiones diferentes. Dado que además el DSS considera las variaciones en las cargas de trabajo, el número de combinaciones posibles aumenta de forma exponencial.

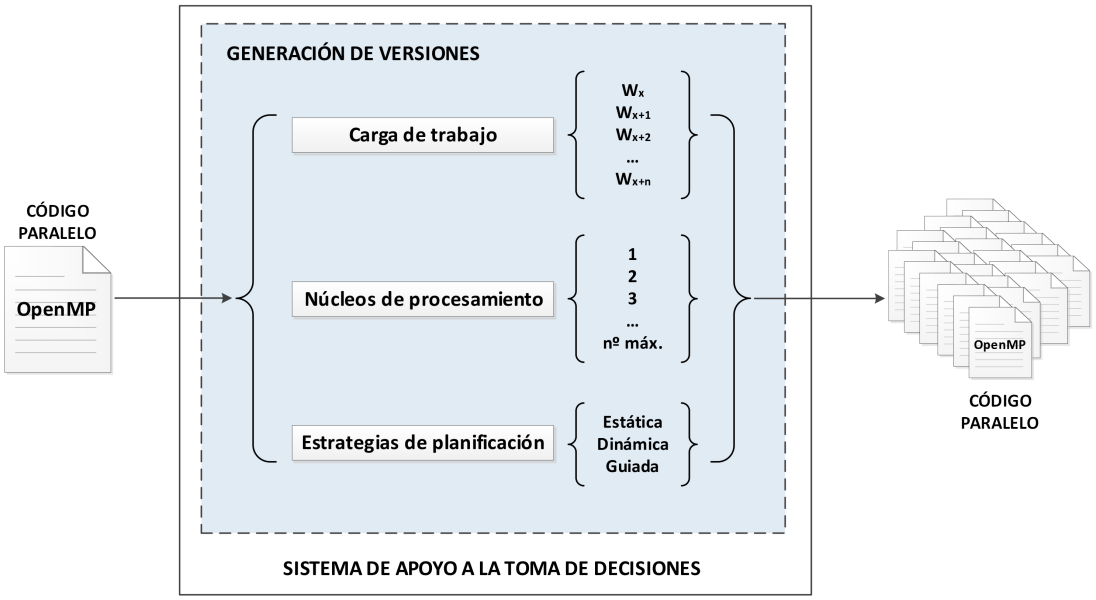

Figura 3.3: Generación de versiones mediante el DSS.

El DSS también permite seleccionar la cola de ejecución de Slurm a la que enviar los trabajos, siendo posible indicar además los nodos específicos que participarán en cada uno, con el objetivo de poder aprovechar sus características particulares.

Tras la finalización de todos los trabajos, el DSS organiza y enumera los resultados para cada programa y carga de trabajo mostrando: los tiempos de ejecución secuenciales y paralelos, el *speedup* (ver apartado 2.2.4.1) o aceleración alcanzada y la eficiencia obtenida. También indica la estrategia de planificación OpenMP que presenta los mejores resultados, el número de núcleos con el que se logra el mayor *speedup*, así como los núcleos necesarios para alcanzar un nivel específico o mínimo de eficiencia. Además, los resultados más destacados son resaltados con objeto de ofrecer una mejor visualización de los datos y agilizar así su interpretación por parte del usuario.

Finalmente, el DSS se encarga de correlacionar las características extraídas de cada bucle paralelizado (descritas en el siguiente apartado) con la planificación OpenMP que obtiene mejor rendimiento para dicho bucle. De esta forma, el DSS también automatiza las tareas de creación y selección de los conjuntos de datos utilizados para entrenar algoritmos de aprendizaje supervisado en el optimizador, facilitando además el análisis por parte de los usuarios de los diferentes factores que influyen en el rendimiento y la eficiencia.

3.2.4.2. Aprendizaje automático

La elección de la planificación paralela más adecuada para ejecutar cada bucle depende de distintos factores, entre los que se incluyen: el tamaño del problema, el número de cores utilizados, la carga del sistema durante la ejecución y las características de la propia infraestructura de cómputo. Escoger una estrategia inadecuada puede tener un

impacto negativo realmente significativo sobre el rendimiento, especialmente en relación al *overhead* asociado a la coordinación de las tareas paralelas.

Por ello, el transcompilador extrae patrones básicos de los códigos a paralelizar y posibilita la implementación de técnicas de aprendizaje automático (ver apartado 2.3.4 sobre el uso de *machine learning* en la computación paralela), para predecir cuál es la mejor planificación OpenMP para ejecutar cada bucle. En concreto, tras una fase inicial de entrenamiento del algoritmo de aprendizaje, el transcompilador utiliza las características de cada bucle a paralelizar y lo clasifica en una de las tres categorías posibles en función de lo aprendido durante el entrenamiento, identificando la planificación más adecuada para su ejecución.

Para desarrollar este módulo se ha empleado WEKA (*Waikato Environment for Knowledge Analysis*) [193, 194], una conocida *suite open source* dedicada a la minería de datos que presenta una variada colección de algoritmos de aprendizaje automático, junto a un extenso número de funciones y herramientas de preprocesamiento y análisis. Su uso facilita la implementación de algoritmos de regresión, clasificación, *clustering* y selección de atributos, entre otros, por lo que es ampliamente utilizada en el ámbito académico [195]. Además, WEKA ofrece varias interfaces gráficas de usuario que posibilitan el acceso a sus funcionalidades implícitas, así como una API basada en Java que ha permitido una perfecta integración en el transcompilador propuesto.

El entrenamiento habilitado emplea aprendizaje supervisado *offline*[10], de modo que los ejemplos están constituidos por las características extraídas de bucles paralelizados que, tras haber sido ejecutados en la infraestructura HPC correspondiente y gracias al análisis posterior realizado por el DSS, incluyen ya en sus directivas OpenMP la planificación con la que se obtienen mejores resultados. Además de este conjunto de entrenamiento, el sistema contempla la utilización de un conjunto adicional de pruebas, exclusivamente para evaluar el rendimiento del algoritmo de aprendizaje. También se posibilita el uso de un tercer conjunto de desarrollo que permite realizar ajustes, seleccionar determinados parámetros o modificar el propio algoritmo.

La extracción de características estáticas (conocidas en tiempo de compilación) de cada bucle paralelo realizada por el transcompilador, permite que cualquier algoritmo de aprendizaje pueda utilizar estos parámetros, por ejemplo, para aprender en caso necesario las ponderaciones que describen la probabilidad de que los patrones de aprendizaje reflejen correctamente las relaciones reales entre cada bucle paralelo y su planificación. En este sentido, existen diversos enfoques de aprendizaje automático que pueden ser empleados con el módulo actual, variando en relación a su coste de computación, complejidad, eficiencia y aplicabilidad. Además, se posibilita la futura ampliación del transcompilador con nuevas funcionalidades de optimización relacionadas con el uso de estos parámetros.

La selección de una caracterización que correlacione adecuadamente los conjuntos de datos con las distintas planificaciones constituye una tarea clave. En este sentido, se han tenido en cuenta todas las variables que pueden influir en el rendimiento de las distintas planificaciones para una ejecución determinada, ya que la elección de estas características

[10]Se considera offline porque se utiliza antes de realizar la ejecución del correspondiente código paralelizado.

tiene un significativo impacto en el modelo predictivo. En concreto, se han considerado las siguientes: niveles de anidamiento (número de bucles anidados); número de iteraciones; número de hilos de ejecución paralelos planificados; operaciones de cómputo realizadas; variables usadas, su tipo (privadas y compartidas) y sus operaciones de lectura/escritura; número de vectores empleados y su tamaño.

Sin embargo, no todas estas características extraídas por el transcompilador (utilizadas en enfoques similares recogidos en la literatura [153, 155, 158] y relacionados previamente en el apartado 2.3.4), presentan el mismo impacto en la clasificación de cada bucle. De hecho, el tamaño del bucle y su carga de trabajo son las particularidades que determinan en mayor grado el tipo de planificación a utilizar [159]. Por ello, se han identificado los siguientes valores como los más importantes:

- Hilos utilizados.
- Iteraciones del bucle.
- Operaciones ejecutadas en su interior.
- Número de niveles de anidación del bucle (que tienen gran impacto en el posible desequilibrio de su carga de trabajo).

El resto de las características han sido descartadas con objeto de evitar que puedan influir de forma negativa en el resultado. No obstante, cabe destacar que existen estrategias de selección que podrían haber sido utilizadas para identificar aquellas características que contienen la información más útil para distinguir entre clases, como el análisis de componentes principales (PCA, *Principal Component Analysis*) [196], la puntuación de información mutua (*mutual information score*) [197], la selección codiciosa [198] o la relación de ganancia de información [199].

Asimismo, dado que los bucles con cargas de trabajo mal balanceadas constituyen uno de los elementos que más influye en el desempeño de cada estrategia de planificación, el transcompilador cuantifica una característica adicional que representa las cargas de trabajo y el desequilibrio de cada bucle, añadiéndola a las ya indicadas. El Algoritmo 4 representa cómo se implementa el cómputo de este valor. El árbol que contiene la información asociada al bucle correspondiente es recorrido en anchura, evaluando las características de dicho bucle y de los posibles bucles anidados internos. De cada uno se extraen el número de iteraciones a realizar y de operaciones de cómputo de las instrucciones que contiene. La carga de trabajo es estimada mediante la multiplicación de ambos valores. En el caso de bucles anidados, la carga del bucle analizado es calculada mediante la multiplicación de las cargas de los bucles de los distintos niveles. Adicionalmente, cuando se produce un desequilibrio de la carga (debido a que la condición de control de un bucle interno depende de la variable de iteración de otro externo), el número de operaciones de dicho bucle es multiplicado nuevamente por la carga de trabajo del mismo. De este modo se garantiza la correcta identificación de aquellos códigos que presentan esta circunstancia y que por tanto, disponen de una carga desequilibrada.

En cualquier caso, en trabajos futuros se prevé ampliar el módulo de optimización considerando también otras características correspondientes a los propios nodos de

Algoritmo 4: Cálculo de la carga de trabajo de un bucle.

```
1  Función cargaTrabajo(Bucle b):
2      desequilibrio ← falso;
3      cargaTrabajo ← 1;
4      para i=0 hasta niveles de anidación del bucle hacer
5          iteraciones ← obtenerNumIteraciones();
6          operaciones ← obtenerNumOperaciones();
7          si (iteraciones distinto de 0) Y (operaciones distinto de 0) entonces
8              cargaTrabajo ← cargaTrabajo × iteraciones × operaciones;
9          si nivel de anidación es mayor que 1 entonces
10             si iteradorBucleExterno es igual a variablesIteraciónBucle entonces
11                 desequilibrio ← verdadero;
12         si desequilibrio entonces
13             cargaTrabajo ← cargaTrabajo × operaciones;
14     devolver cargaTrabajo;
```

cómputo donde se realizan las ejecuciones, como por ejemplo, la carga del sistema en cada momento o las propiedades de su infraestructura, dado el impacto que las particularidades de cada equipamiento puede presentar en los resultados. De igual modo, se podrán considerar parámetros adicionales de los bucles, valorando su relevancia y evaluando su impacto en el rendimiento final de cada esquema de planificación, como por ejemplo, la utilización de estructuras condicionales o las llamadas a funciones.

Actualmente el transcompilador reconoce los patrones en los bucles y clasifica sus ejecuciones en los tres tipos de planificación OpenMP mediante el algoritmo de los k vecinos más próximos (*k-nn*, *k-nearest neighbors*) [160], que aproxima la función de predicción de forma local y pospone el cómputo a la fase de clasificación.

De este modo, las características extraídas para cada bucle operan como vectores de un espacio característico multidimensional (una dimensión por cada característica), donde cada uno es descrito en términos de los 5 atributos considerados (hilos de ejecución, iteraciones del bucle, operaciones realizadas, niveles de anidación y carga de trabajo) y se consideran las tres estrategias de planificación posibles. Por tanto, los valores del i-ésimo bucle del conjunto de entrenamiento X son representados de la siguiente forma:

$$x_i = (x_{i1}, x_{i2}, x_{i3}, x_{i4}, x_{i5}) \in X. \tag{3.6}$$

Durante la fase de entrenamiento, los vectores correspondientes a los bucles paralelizados son asociados con la planificación que obtiene mejor *speedup* y son añadidos a la base de datos mediante la utilización del DSS, que previamente ha seleccionado dicha planificación para este bucle, a partir de los resultados de las distintas ejecuciones asociadas al mismo. Por tanto, el algoritmo de entrenamiento considera cada bucle como:

$$\langle x, f(x) \rangle \tag{3.7}$$

con $x \in X$. De esta forma, para construir el modelo de predicción, cada bucle de entrenamiento es representado por un vector que es almacenado con las características del mismo y una etiqueta que indica su planificación más adecuada.

Cuando el transcompilador requiere paralelizar un nuevo bucle, determina en primer lugar las directivas *pragma OpenMP* correspondientes y extrae sus propiedades, incluyéndolas en el correspondiente vector y representándolas en el espacio de características. De este modo, el bucle es clasificado en una de las tres estrategias de planificación OpenMP cuando dicha estrategia es la más frecuente en los k puntos del espacio más cercanos, utilizando para ello la distancia euclídea:

$$d(x_i, x_j) = \sqrt{\sum_{r=1}^{n}(x_{ir} - x_{jr})^2} \tag{3.8}$$

siendo x_{ir} y x_{jr} los atributos r-ésimos de x_i y x_j respectivamente y n el número de dimensiones. Es decir, tras calcular la separación entre los vectores almacenados en el conjunto de entrenamiento y el nuevo vector, éste último es clasificado en la planificación que más apariciones presenta entre los más cercanos. Por tanto, el algoritmo de clasificación considera los k vecinos del conjunto de entrenamiento más próximos al bucle x_q a categorizar de la siguiente forma:

$$\hat{f}(x) \leftarrow \arg\max_{v \in V} \sum_{i=1}^{k} [v = f(x_i)] \tag{3.9}$$

con $v \in V$, siendo V el conjunto de posibles planificaciones (estática, dinámica y guiada) y $\hat{f}(x)$ el valor más común de f entre los k vecinos más cercanos al nuevo vector (ver Algoritmo 5). Considerando k=*1*, se restringe la clasificación a la planificación establecida por únicamente el vecino más cercano, sin tener en cuenta otros. En general, el valor de k puede ser ajustado automáticamente mediante una función *fit* que utiliza como parámetro de entrada el conjunto de entrenamiento.

En futuras líneas de desarrollo se prevé utilizar los datos que extrae automáticamente el DSS relativos a cada ejecución, de forma que el *speedup* y la eficiencia del código pasen a formar parte de las características utilizadas por el optimizador para realizar las predicciones, permitiendo correlacionar así los parámetros del bucle y el número de cores utilizados, con el rendimiento paralelo obtenido. De este modo, será posible predecir la planificación que obtiene el mejor *speedup* con una eficiencia mínima o encontrar un equilibrio entre ambos parámetros, con objeto de mejorar el aprovechamiento de los recursos de cómputo disponibles. Asimismo, también se prevé ampliar el alcance actual considerando el tamaño de los *chunks* utilizados en las distintas planificaciones.

Algoritmo 5: Clasificador de planificaciones OpenMP mediante los k-vecinos más cercanos

```
1 Función clasificador(conjuntos de entrenamiento ⟨X, f(X)⟩, bucle a clasificar x_q):
2     para cada ejemplo de entrenamiento ⟨x, f(x)⟩ hacer
3         añadirAConjuntoEntrenamiento(x, f(x));
4     devolver f̂(x_q) ← arg máx_{v∈V} Σ_{i=1}^{k} [v = f(x_i)];
5     siendo x_i a x_k los k puntos del conjunto de entrenamiento más cercanos a cada atributo de x_q.
6     siendo V = estática, dinámica, guiada.
7     siendo f̂(x) el valor más común de f entre los k ejemplos.
```

Además, se estudiará la limitación del número de versiones similares generadas por el DSS, con objeto de disminuir el número de ejecuciones iniciales necesarias sin perder la representatividad de las muestras analizadas. Actualmente, las ejecuciones asociadas a los conjuntos de entrenamiento creados con el DSS suponen un alto coste computacional que puede resultar inaceptable en ciertos escenarios HPC. Para ello se plantea la utilización de heurísticas que permitan orientar mejor las optimizaciones.

Capítulo 4

Técnicas para el desarrollo de códigos de programación eficientes

El presente capítulo proporciona una descripción pormenorizada de las veintiséis técnicas *software* propuestas en esta investigación para la escritura de códigos de programación eficientes. Se plantea de este modo un conjunto de estrategias y pautas que permiten a los programadores alcanzar mejoras realmente significativas en sus códigos, mediante la aplicación de una serie de pequeñas modificaciones y empleando un esfuerzo mínimo.

La utilización de las técnicas permite complementar las ventajas ofrecidas por el transcompilador, especialmente en relación a las partes secuenciales de un programa que no pueden ser paralelizadas. El objetivo es mejorar el rendimiento, consiguiendo reducir lo máximo posible los tiempos de ejecución y el consumo energético en los centros de cómputo. En este sentido, es necesario considerar el destacable efecto negativo que simples fragmentos de código pueden presentar sobre los tiempos de ejecución, debido a la importante limitación que la fracción secuencial de un programa ejerce sobre el *speedup* final alcanzado. De este modo, su aplicación permitirá aumentar las ventajas obtenidas con las paralelizaciones automáticas realizadas por el transcompilador.

Asimismo, se pretende que los programadores comprendan el importante impacto que la utilización de ciertas técnicas puede tener en el rendimiento final de sus aplicaciones, especialmente en aquellas que están en continua ejecución o bien consumen un número considerablemente elevado de horas de CPU. En este sentido, algunos programadores no son conscientes, por ejemplo, de la significativa cantidad de tiempo que puede llegar a ser empleada por un pequeño fragmento de código ineficiente, especialmente cuando la ejecución de éste es repetida en numerosas ocasiones.

No obstante, cabe mencionar que, en ciertos casos, la aceleración de los programas para mejorar su eficiencia mediante la aplicación de estas técnicas, puede conllevar incrementos en el tamaño del código que afecten a su legibilidad. Este factor no ha sido tenido en cuenta durante la selección de las distintas técnicas, priorizando siempre su nivel de efectividad en la reducción de los tiempos de ejecución y los consumos de energía.

Todas las técnicas presentadas a continuación han sido seleccionadas tras efectuar una extensa revisión de la literatura científica existente, escogiendo aquellas más representativas o que mejoran de forma más notable la eficiencia del código.

4.1. Técnicas de programación eficientes

A continuación, se describen las técnicas de programación eficientes que analiza y evalúa la presente tesis doctoral [171–180], las cuales pueden ser aplicadas recurrentemente, afectando de manera significativa al rendimiento del código:

T1 ***Campos de bits***: los *bit fields* o campos de *bits*, son estructuras de datos empleadas para contener secuencias de un cierto número de *bits* con el objetivo de optimizar el espacio de memoria requerido. Su uso sustituye la utilización de otros tipos de datos donde dicho espacio solo se aprovecha parcialmente, por ejemplo, requiriendo 8 *bits* y solo utilizando 2. Esto es común en variables con solo dos estados posibles, como las de tipo "verdadero o falso". Sin embargo, estos campos suelen ser accedidos mediante punteros, requiriendo accesos individuales para cada uno de los *bits* y originando, en ocasiones, problemas de *aliasing*[1]. La presente técnica recomienda remplazar el uso de *bit fields* por números enteros segmentados mediante la aplicación de máscaras `enum` o `#define`, lo cual posibilita obtener todos los *bits* mediante un único acceso.

T2 ***Conjuntos de bits***: similares a los *bit fields* descritos en la técnica anterior, los conjuntos de *bits* o *bitsets* almacenan elementos que requieren solo dos valores posibles y emulan el funcionamiento de los vectores de booleanos, pero a diferencia de estos, sus valores no se almacenan de forma separada. De este modo, optimizan el espacio asignado y los accesos son más rápidos que mediante la utilización de vectores dinámicos. Además, permiten el uso de operadores lógicos, pueden ser convertidos a (y desde) valores enteros o cadenas binarias y es posible inicializarlos con una única instrucción: `bitset.set()`. Así, esta técnica recomienda priorizar su uso en situaciones en las que se requieran vectores booleanos, sustituyéndolos por conjuntos de *bits* con el mismo número de elementos. No obstante, es importante tener en cuenta que dado que los *bitsets* representan secuencias de *bits* de tamaño fijo, este tamaño debe ser establecido en tiempo de compilación, a diferencia de matrices y vectores dinámicos cuyas dimensiones pueden cambiar durante la ejecución.

T3 ***Retorno de booleanos***: cuando una función utiliza diversas variables de tipo booleano, es posible transformar todas estas variables en un único entero sin signo (*unsigned int*) de 16 *bits*, mediante la definición de banderas asociadas. Además, esto posibilita realizar diferentes comprobaciones a través de una única operación lógica y permite, por ejemplo, devolver ocho variables booleanas a través de la definición de sus ocho banderas correspondientes, (`flagX (1u <<` x`)`), siendo `X={A,B,..,H}` y `0` $\leq x \leq$ `7`) y utilizando para ello una única operación de retorno, de forma que la secuencia de booleanos "`0,0,0,0,1,1,0,0`" podría ser devuelta por la función con una única variable *unsigned int* con valor `12`, equivalente en binario a "`00001100`".

[1] Situación en la cual es posible acceder a una única dirección de memoria desde dos punteros diferentes.

T4 ***Llamadas en cascada a funciones***: en la medida de lo posible, se deben evitar las llamadas en cascada a funciones que retornen punteros o referencias, especialmente si se encuentran dentro de bucles que llegan a implicar un alto número de repeticiones. En aquellos casos en los que la referencia devuelta por la función permanezca sin cambios durante las distintas llamadas a la misma, se recomienda almacenar previamente su valor de retorno en una variable auxiliar y utilizar ésta en sustitución de las llamadas a la función. En este caso, el programador juega un rol especialmente significativo, dado que la ventaja de aplicar esta técnica viene dada principalmente por el hecho de que él, a diferencia del compilador, puede ser consciente de que la referencia devuelta por una función determinada puede mantenerse constante durante la ejecución de un fragmento concreto de código.

T5 ***Acceso por fila principal en matrices***: lenguajes como Fortran o MATLAB realizan su almacenamiento en orden contiguo por columna principal, mientras que en C y C++ son implementadas mediante orden de fila principal. De este modo, considerando la siguiente matriz:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

en Fortran, los elementos de cada columna son consecutivos en memoria, siendo almacenados en el siguiente órden: `A = [1,4,7,2,5,8,3,6,9];` y en cambio en C y C++ los elementos de cada fila se almacenan en posiciones consecutivas de la siguiente forma: `A = [1,2,3,4,5,6,7,8,9]`. Es decir, el elemento `[x + 1]` es almacenado de forma contigua a `[x]`, o dicho de otro modo, considerando un vector `[i][j]`, el subíndice `[j]` es el que varía más a menudo durante el recorrido del vector. Por ello, esta técnica sugiere incrementar en primer lugar el índice más a la izquierda al recorrer matrices bidimensionales, con el objetivo de lograr una mayor tasa de aciertos de caché al acceder a cada elemento. Es decir, considerando el vector anterior, éste debe ser siempre recorrido mediante dos bucles *for*: uno exterior que itere sobre la variable "`i`", anidado a otro interior que en este caso itere sobre la variable "`j`".

T6 ***Listas de inicialización***: se recomienda utilizar listas de inicialización para establecer los valores iniciales de los constructores y evitar así las sobrecargas innecesarias generadas por las asignaciones posteriores. De este modo, se debe eludir la invocación de estas variables únicamente para inicializar sus valores en el cuerpo del constructor, puesto que cada una de estas asignaciones implica, a su vez, una invocación al constructor predeterminado de la variable correspondiente. Así, instrucciones de inicialización como "`attribute1 = value1`", siendo `attribute1` de tipo `string`, implican, además de la llamada al constructor `string` del atributo, otra llamada al operador de asignación (`=`) de `string` para copiar el valor, así como una llamada adicional al destructor `string` cuando el atributo queda fuera del ámbito donde se realiza la asignación. En su lugar, esta técnica propone la inicialización de todas los atributos del constructor mediante una lista, estableciendo de forma directa sus valores y utilizando, por tanto, una única vez el constructor de cada atributo. Por ejemplo, `ClassConstructor (typeAtt1 value1, typeAtt2 value2, [...]) : attribute1(value1), atribute2(value2), [...])`, siendo `typeAttX` el tipo del atributo `X` y `valueX` su valor de inicialización.

T7 ***Eliminación de subexpresiones comunes***: esta técnica consiste en evitar el uso de expresiones idénticas que son ejecutadas reiteradamente, sustituyéndolas por una única variable que almacena el resultado de sus operaciones y que reemplaza en el resto del código a la expresión completa. De este modo, aquellas expresiones idénticas que son calculadas repetida e innecesariamente debido a que los valores de sus operandos no cambian, pueden ser reemplazadas por una variable que contenga el resultado de estos cálculos.

T8 ***Mapeo de estructuras***: al realizar búsquedas en tablas de mapeo de valores a constantes, se recomienda recorrer cada tabla mediante un bucle hasta encontrar la constante correcta, en lugar de utilizar sentencias condicionales `if` anidadas que comprueben cada uno de los casos. Las tablas de mapeo son estructuras ordenadas que, siguiendo un orden específico, almacenan elementos consistentes en pares clave-valor formados por un valor mapeado y la clave que lo identifica de manera única. Además, estas tablas suelen ser especialmente recomendables cuando se realizan pocas inserciones. Por el contrario, el uso de las sentencias `if` anidadas provoca que el compilador traduzca cada caso individualmente, lo cual conduce a un peor rendimiento por norma general.

T9 ***Eliminación de código muerto***: esta técnica es una de las optimizaciones más conocidas y ampliamente utilizadas por los compiladores. Consiste en eliminar instrucciones que son totalmente inalcanzables durante la ejecución del código o bien no influyen en el resultado del mismo (por ejemplo, en el caso de variables que son definidas pero posteriormente nunca son utilizadas).

T10 ***Control de excepciones***: las excepciones permiten gestionar errores inesperados cuando se producen condiciones anómalas o particulares que originan cambios en el flujo normal de las ejecuciones. Por ello, todos los contenedores de la librería estándar de C ++ utilizan expresiones que arrojan excepciones [174]. Sin embargo, dado que su uso también implica importantes penalizaciones de tiempo, es posible mejorar notablemente el rendimiento en ciertas situaciones en las que las excepciones son utilizadas dentro de bucles, ya que en algunos casos, éstas pueden ser remplazadas por otras instrucciones. Por ejemplo, es posible detectar y manejar errores mediante sentencias como `continue` y `break`, que pueden ofrecer la misma funcionalidad sin tener que lanzar excepciones, reduciendo así considerablemente los tiempos de ejecución. En este sentido, se debe tener en cuenta que, incluso excepciones simples sin ningún contenido salvo su declaración (por ejemplo, `class myexception: public exception {} myex;`), originan sobrecargas adicionales de tiempo. Esto es debido a que las llamadas realizadas al manejador de excepciones implican penalizaciones, dado que siempre es necesario pasarle un parámetro como argumento, cuyo tipo es comparado con el tipo de parámetro definido por el manejador. Cuando ambos coinciden, se detecta la excepción y el control del flujo de ejecución es transferido al controlador. Por lo tanto, es especialmente recomendable utilizar esta técnica en aquellos bucles que presentan una alta probabilidad de existencia de excepciones, pero éstas pueden ser gestionadas sin tener que ser arrojadas. Su grado de mejora dependerá directamente del número de ellas lanzadas y de la estructura de datos empleada.

T11 ***Variables globales en bucles***: dentro de bucles, las asignaciones a variables globales deben ser evitadas en la medida de lo posible. Éstas pueden ser sustituidas por

variables locales en las instrucciones requeridas, de modo que tras finalizar el bucle, se realice una única asignación a cada variable global.

T12 ***Funciones inline o insertadas***: en muchos casos, las sobrecargas originadas por llamadas a funciones y sus correspondientes retornos pueden ser evitadas insertando directamente el código completo contenido por estas funciones en lugar de llamarlas. Esta técnica supone, por tanto, un método efectivo para eliminar el *overhead* causado cada vez que se produce una llamada a una función. De hecho, algunos autores la señalan como la mejor optimización [174]. Además, tiene destacables ventajas frente al uso alternativo de macros *#define*, dado que estos no verifican los tipos de función [172]. En general, esta técnica consiste en reemplazar la llamada a una función, introduciendo en su lugar directamente el código completo de la misma, a excepción de las sentencias de retorno.

T13 ***Variables globales***: en general, se recomienda evitar el uso intensivo de variables globales debido a su sobrecarga asociada. Se trata de variables definidas fuera del cuerpo de cualquier función y por tanto, accesibles desde todos los ámbitos del código. En primer lugar, esto puede dificultar la tarea del programador, que debe razonar todas las posibilidades de cada variable global durante el desarrollo, pudiendo llegar a olvidar fácilmente cualquier regla relacionada con su uso. De este modo, el hecho de que estas variables puedan ser modificadas por cualquier función, puede originar importantes problemas colaterales, al provocar que varias funciones interfieran entre sí y alteren sus valores. Además, dado que las variables globales no son almacenadas directamente en registros, estas no deben ser utilizadas dentro de bucles críticos para evitar *overheads* innecesarios provocados por el acceso reiterativo a su contenido en memoria. En consecuencia, respecto al tiempo de ejecución, esta técnica recomienda que cuando una función presente un uso intensivo de variables globales, se copien previamente sus valores en variables locales y se utilicen éstas en su lugar, mejorando así los tiempos de acceso.

T14 ***Constantes en bucles***: se propone evitar, cuando sea factible, accesos continuos a constantes en sentencias ubicadas en el interior de bucles, aunque en ocasiones esto conduzca a una peor legibilidad del código.

T15 ***Inicialización frente a asignación***: independientemente del tipo de datos utilizado, se sugiere inicializar directamente las variables al ser declaradas, en lugar de establecer su valor en otra sentencia. De este modo se evita la asignación adicional provocada por su instanciación posterior.

T16 ***División con denominadores potencias de 2***: esta técnica consiste en modificar expresiones de división, cuyos denominadores son potencias de 2, mediante operadores de desplazamiento lógicos *bit* a *bit* que desplazan los *bits* hacia la derecha un número de posiciones especificado con el operador “>>” [200]. Las ubicaciones que quedan vacías son completadas con ceros. De este modo, el valor de “`a >> b`” sería el valor de “`a`” desplazado a la derecha “`b`” posiciones de *bits*.

T17 ***Multiplicación con factores potencias de 2***: similar a la anterior, esta técnica consiste en reemplazar expresiones de multiplicación cuyos factores son potencias de 2, empleando también para ello operadores de desplazamiento lógico, en este caso utilizando el operador “<<”.

T18 ***Integer frente a character***: para realizar operaciones aritméticas es preferible utilizar `Integer` en lugar de `char`, dado que en el caso del segundo tipo, C y C++ convierten los valores de caracteres a enteros antes de operar, para después volver a convertir el resultado de nuevo al tipo `char` original, lo cual puede suponer una sobrecarga innecesaria.

T19 ***Bucles con cuenta regresiva***: cuando el orden seguido por el iterador de un bucle no es determinante, es posible aplicar esta técnica consistente en recorrer el bucle hacia atrás, en sentido opuesto. De este modo se aprovecha el hecho de que es más rápido procesar "*i--*" como condición del bucle. Atravesar un bucle hacia delante (por ejemplo, con condición `i=0; i<100`) requiere más instrucciones: restar "`i`" a 100, evaluar si el resultado es cero, si no, incrementar el iterador y continuar. Esto puede implicar una significativa variación de tiempo, especialmente en bucles con un amplio número de repeticiones. La diferencia se incrementa cuando además cambiamos el bucle de tipo `for` a tipo `while`, mediante una sentencia cuya única condición sea "`mientras --i hacer`". También es recomendable reducir, en la medida de lo posible, el coste de la cláusula de terminación de cada bucle, dado el número de ocasiones que pueden llegar a ser evaluadas.

T20 ***Desenrollado de bucles***: desenrollar pequeños bucles puede conllevar significativas reducciones en sus tiempos de ejecución, principalmente por reducirse el número de iteraciones realizadas y, en consecuencia, el número de ocasiones que se ejecutan las instrucciones que controlan los bucles en cada iteración, las cuales pueden implicar importantes penalizaciones de tiempo. El desenrollado consiste en disminuir el número de iteraciones a realizar y reescribir el cuerpo del bucle, repitiendo en él tantas secuencias de instrucciones independientes similares como sea necesario (en función del número de iteraciones eliminadas). Esta técnica presenta la desventaja de aumentar el tamaño del código, por lo cual debe ser aplicada de forma prudente y únicamente en bucles sin un número elevado de repeticiones. De hecho, el desenrollado manual o estático realizado por el programador implica analizar previamente el bucle para interpretar sus iteraciones, a diferencia del desenrollado dinámico realizado por el compilador. En consecuencia, con objeto de evitar que el rendimiento de la caché pueda verse afectado, únicamente se recomienda desenrollar aquellos bucles que realmente afecten al tiempo de ejecución final.

T21 ***Paso de estructuras por referencia***: siempre que sea posible, cuando ser realizan llamadas a funciones, se recomienda pasar las estructuras por referencia, a fin de evitar la significativa sobrecarga en la que se incurre cuando son pasadas por valor, puesto que esto implica la realización de una copia completa de cada estructura (incluyendo los correspondientes métodos constructor y destructor de las mismas). Dado que los argumentos referenciados pueden modificar la instancia original (a diferencia de los argumentos por valor), se sugiere que, en estos casos, los punteros a las estructuras sean declarados como constantes, siempre que se tenga constancia de que los valores apuntados no van a ser alterados.

T22 ***Aliasing de punteros***: en esta técnica, el término *aliasing* hace mención a situaciones en las cuales dos o más punteros apuntan a la misma dirección (o variable) y, por tanto, la escritura de uno puede afectar a la lectura de otro. Esta posibilidad impide en ocasiones que el compilador pueda aplicar ciertas optimizaciones, implicando reducciones en la eficiencia. De este modo el compilador

no eliminaría automáticamente, por ejemplo, varias subexpresiones comunes (técnica T7, mencionada anteriormente) que incluyesen el uso de un mismo puntero.

T23 ***Cadenas de punteros***: esta técnica, directamente relacionada con la anterior, consiste en optimizar una cadena de punteros (un puntero apunta a otro puntero y así sucesivamente: `p1->p2->p3->`...) cuando ésta es utilizada repetidamente para acceder a un determinado valor. En concreto, se propone utilizar variables locales intermedias para acceder directamente a los eslabones finales (por ejemplo, `*aux = p1->p2->p3`), de forma que posteriormente sea posible acceder al resto de la secuencia a través de `aux` sin tener que recorrer la cadena completa para llegar a `p3`.

T24 ***Pre-incremento frente a pos-incremento***: se recomienda priorizar el uso de operadores de pre-incremento en tipos de datos cuyas clases sobrecargan ambos, dado que el operador de incremento posterior crea previamente una copia adicional del objeto que no es necesaria si se utiliza el pre-incremento.

T25 ***Búsqueda lineal***: aunque existen otros algoritmos de búsqueda que ofrecen resultados similares en menor tiempo, la búsqueda lineal es especialmente utilizada en listas con pocos elementos, o bien con fines de aprendizaje por parte de programadores noveles. Generalmente, una búsqueda lineal para encontrar un elemento dentro de una lista mediante un bucle `for` implica dos comparaciones principales: una para controlar las iteraciones del bucle y otra para verificar si el elemento actual coincide con el valor buscado. Esta técnica permite mejorar el rendimiento de la búsqueda, reemplazando el bucle `for` por una estructura de tipo `while`, que únicamente requiere una comparación (`while (lista[i] != buscado)`), para controlar el flujo de ejecución. Con objeto de evitar el problema que surgiría en caso de que la lista no contuviese el valor buscado, es necesario insertar previamente dicho valor al final del vector (aumentando su tamaño en un elemento). Además, de esta forma también es posible averiguar si el valor existe en la lista simplemente verificando si el índice encontrado se corresponde con la última posición.

T26 ***Estructuras If en bucles***: esta técnica consiste en transformar un único bucle en dos bucles distintos, cuando el original contiene una estructura condicional `if` que no varía una vez comenzada su ejecución. Para ello, se extrae la condición que será ejecutada en primer lugar y, a continuación, se generan dos bucles diferentes, correspondientes a cada condición del `if`.

Estas veintiséis técnicas han sido analizadas en profundidad, realizando una evaluación pormenorizada de cada una y desarrollando diversas implementaciones y experimentos *ad hoc* para demostrar su eficiencia. El impacto de su utilización sobre varios procesadores y entornos es descrito de forma detallada en el Capítulo 5. En él se recogen los resultados experimentales de las distintas pruebas realizadas y se expone la eficiencia de cada técnica para alcanzar destacables reducciones en el tiempo de ejecución y en el consumo energético. Además, se demuestra que el programador ejerce un gran impacto en la eficiencia del código y su influencia sigue siendo decisiva. Así, gracias a la aplicación de estas técnicas, es posible mejorar substancialmente el rendimiento obtenido mediante las optimizaciones automáticas ofrecidas por los compiladores de código.

Capítulo 5

Resultados experimentales

En este capítulo se presentan y analizan los resultados experimentales obtenidos en esta tesis: en primer lugar en el apartado 5.1, en relación al transcompilador presentado en el Capítulo 3, analizando la paralelización automática de códigos secuenciales; y a continuación en el punto 5.2, respecto a la aplicación de las técnicas introducidas en el Capítulo 4, para el desarrollo de códigos eficientes en dispositivos IoT e infraestructuras de cómputo de alto rendimiento.

5.1. Aplicación del transcompilador a la paralelización automática

En este primer apartado se analiza la capacidad del transcompilador en cuanto a la paralelización automática de códigos secuenciales. Para ello se utiliza un conjunto de nueve problemas, escogidos por su representatividad en diversos ámbitos, que pueden ser resueltos mediante la utilización de la programación paralela, alcanzando de esta forma unos tiempos de ejecución ampliamente inferiores a los obtenidos mediante sus versiones secuenciales. A continuación, el apartado 5.1.1 presenta la metodología seguida así como los distintos experimentos desarrollados. Posteriormente, en el punto 5.1.2 se analizan y evalúan los resultados obtenidos.

5.1.1. Metodología y experimentos

Los experimentos desarrollados para analizar la paralelización automática ofrecida por el transcompilador han sido ejecutados sobre dos infraestructuras distintas. En los primeros resultados publicados [201] se utilizó un servidor Fujitsu Primergy RX4770 M2 con 4 procesadores Intel Xeon E7-4830v3, correspondiente a un nodo de cómputo de memoria compartida. En la siguiente publicación [202] se empleó otro servidor Fujitsu, en este caso un CX2550 con 2 procesadores Intel Xeon E5-2660v3, pertenecientes a un clúster de memoria distribuida. No obstante, para la redacción del presente documento, se han

ejecutado nuevamente todos los experimentos en ambos sistemas, con objeto de poder ofrecer una comparativa más completa y detallada de los resultados obtenidos.

Las características de los dos sistemas empleados son mostradas en la Tabla 5.1 [203]. En adelante, por razones de legibilidad se hará referencia a sus procesadores (E7-4830v3 y E5-2660v3) para diferenciar ambos nodos de cómputo. En este sentido es necesario considerar que las paralelizaciones automáticas realizadas por el transcompilador son desarrolladas en OpenMP y están destinadas principalmente a sistemas de memoria compartida (por las razones expuestas en los Capítulos 2 y 3). Por ello, como puede observarse en la tabla, los experimentos desarrollados en el sistema de memoria compartida emplean 4 procesadores E7-4830v3 (con 48 cores en total), mientras que los de memoria distribuida, pese a ejecutarse en un clúster, únicamente utilizan un nodo con 2 procesadores E5-2660v3 (20 cores entre ambos) que comparten una memoria de 80 GB.

Asimismo, existen significativas diferencias entre ambos sistemas, más allá de las características inherentes a la utilización de sistemas de memoria compartida o distribuida (detalladas en el apartado 2.1 del Capítulo 2). Principalmente, en relación a las ventajas del E7-4830v3, éste presenta el doble de *sockets* por nodo (4 frente a los 2 procesadores E5-2660v3). Este incremento posibilita obtener mejoras significativas en relación a su escalabilidad, permitiendo realizar ejecuciones con un mayor número de procesos o tareas paralelas. En concreto, este *fat node* de memoria compartida dispone de un total de 48 cores. Por otro lado, los procesadores E5-2660v3 del nodo de cómputo de memoria distribuida presentan 0,50 GHz más de frecuencia base, lo que se traduce en un incremento de velocidad del 23,80 %. Esto origina que, en general, cuando se utiliza el mismo número de cores, las ejecuciones sean más rápidas en este sistema. Sin embargo, su memoria (aunque de mayor frecuencia: 2133 MHz frente a 1333 MHz) está limitada a los 80 GB del nodo, frente al 1,5 TB total disponible en el de memoria compartida. Cabe mencionar sin embargo, que su organización *smart cache* permite que todos los cores tengan acceso a la caché de último nivel (L3), mejorando la utilización de los recursos y permitiendo obtener datos en caché L2 incluso antes de que estos sean solicitados [203].

Ambos sistemas han sido escogidos para realizar los experimentos por su representatividad en el ámbito HPC y por poseer unas características ampliamente utilizadas en los centros de supercomputación actuales. Además, considerando el uso del transcompilador en CénitS, los nodos de cómputo que presentan estos procesadores

Tabla 5.1: Especificaciones de los procesadores donde se han desarrollado los experimentos sobre paralelización automática.

	Intel Xeon E7-4830v3	**Intel Xeon E5-2660v3**
Nodo de cómputo	Memoria compartida	Memoria distribuida
Nº de Procesadores	4 (E7-4830v3)	2 (E5-2660v3)
Nombre clave proc.	Haswell	Haswell
Nº de Cores	12 (por procesador)	10 (por procesador)
Nº total de Cores	48 cores (por nodo)	20 cores (por nodo)
Frecuencia base	2,10 GHz	2,60 GHz
Caché	30 MB LLC (Last Level Cache)	25 MB SmartCache
Memoria	1,5 TB DDR4 @ 1333 MHz	16 GB DDR4 @ 2133 MHz (80 GB por nodo)
Disco duro local	Fujitsu PRAID CP400i SAS 300 GB 12 Gbps	Innodisk SATADOM-MH 3ME 120 GB 6Gbps

destacan entre los más solicitados por los investigadores y usuarios del centro para ejecutar sus códigos, formando parte de su infraestructura de cómputo [204]. En cualquier caso, la funcionalidad y ventajas obtenidas en los resultados de las pruebas realizadas serían semejantes en otro tipo de recursos HPC. Asimismo, se prevé ampliar esta investigación incluyendo otras infraestructuras y arquitecturas computacionales, de forma que los resultados ofrecidos puedan ser extrapolados de forma más exacta a un mayor número de centros de supercomputación.

Para analizar la paralelización automática ofrecida por el transcompilador se han desarrollado una serie de experimentos relacionados con distintos problemas, los cuales han sido escogidos por su potencial paralelismo y su amplia utilidad en diversos ámbitos. Para realizar las mediciones de tiempo de estos experimentos se ha utilizado *GNU time* [205] (empleando directamente el binario "`/usr/bin/time`", en lugar de la versión integrada en la *shell*), al presentar una resolución apropiada en relación a los tiempos de ejecución manejados. A continuación se detallan estos experimentos.

5.1.1.1. Conjunto de Mandelbrot

El primer experimento consiste en la paralelización de un programa secuencial que genera un conjunto de Mandelbrot [206, 207], uno de los fractales[1] más conocidos. Se trata de un conjunto de números complejos c, es decir de puntos del plano, para los que la sucesión obtenida mediante la siguiente ecuación de recurrencia cuadrática no diverge:

$$\begin{cases} Z_0 = 0 \in \mathbb{C} \\ Z_{n+1} = {Z_n}^2 + c \end{cases} \tag{5.1}$$

Es decir, cuando la sucesión recursiva $Z_{n+1} = (x_n, y_n)$ permanece acotada en valor absoluto. Por tanto, cada sucesión de puntos (x_n, y_n) puede ser generada a partir de $c = (x, y)$ mediante las siguientes ecuaciones:

$$x_{n+1} = {x_n}^2 - {y_n}^2 + x \tag{5.2}$$

$$y_{n+1} = 2 \cdot x_n \cdot y_n + y \tag{5.3}$$

La paralelización es posible debido a que cada punto puede ser analizado de forma independiente al resto. De este modo, para realizar el gráfico del fractal, el algoritmo comienza en un punto inicial y calcula la sucesión (ecuación 5.2) para un número concreto de pasos, habiendo considerado en este experimento $2 \cdot 10^6$ repeticiones. Si sus resultados nunca exceden una magnitud de radio 2, el punto se considera miembro del conjunto de Mandelbrot.

El Código B.1 (recogido en el Apéndice B, -página 134-) muestra el bucle que realiza los cálculos necesarios para evaluar la pertenencia de cada pixel al conjunto. En él puede observarse la directiva *pragma* generada automáticamente por el transcompilador (ver línea 1 del código), donde se identifican y clasifican las distintas variables como privadas

[1] Un fractal es un objeto geométrico con una estructura básica que es repetida a distintas escalas en base a ecuaciones matemáticas complejas. Asimismo, la apariencia y distribución estadística de sus estructuras no varía aunque se modifique la escala de observación [206].

(línea 2) y compartidas (línea 3), ofreciendo una noción de la complejidad que podría suponer su programación manual para usuarios noveles en paralelismo.

5.1.1.2. Cálculo de números primos

Programa que calcula y muestra los números primos existentes entre 1 y 10^{10}. El código B.2 (pág. 135) muestra el bucle principal implementado y paralelizado automáticamente. En este caso, el transcompilador identifica además dos operaciones de reducción en la suma ejecutadas sobre dos variables (ver líneas 4 y 5 del código).

5.1.1.3. Estimación de integral $\int_a^b f(x)\,\mathrm{d}x$

Programa que realiza una estimación de la integral

$$\int_a^b f(x) = \frac{50}{\pi \cdot (2{,}5 \cdot 10^2 \cdot x^2 + 1)}\,\mathrm{d}x \tag{5.4}$$

para $a = 0$ y $b = 10$, utilizando un algoritmo de integración numérica de 10^{11} nodos[2] ($N = 10^{11}$) (ver Código B.3, -pág. 135-). En concreto, el algoritmo se aproxima al resultado de la integral mediante la ecuación:

$$|A|\frac{1}{N}\sum_{i=1}^{N} f(x_i) \tag{5.5}$$

siendo $|A| = (b-a)$ el volumen del dominio de la integración y x_i una serie de números distribuidos uniformemente en el rango $a \leq x_i \leq b$ de forma que:

$$x_i = \frac{(N-i-1)\cdot a \;+\; i \cdot b}{(N-1)} \tag{5.6}$$

5.1.1.4. Estimación de integral doble $\int_R f(x,y)\,\mathrm{d}x\mathrm{d}y$

Programa que ejecuta una estimación de la integral doble

$$\int_R f(x,y) = \frac{1}{(1-x\cdot y)}\,\mathrm{d}x\mathrm{d}y \tag{5.7}$$

sobre el rectángulo $R = [0,1]x[0,1]$, utilizando un algoritmo de integración numérica de $m \times n = 3 \cdot 10^5 \times 3 \cdot 10^5$ nodos (ver Código B.4, -pág. 136-). La aproximación del algoritmo al resultado de la integral se realiza utilizando la ecuación:

$$\frac{(b-a)^2}{m \times n} \cdot \sum_{i=1}^{m\times n} f(x_i, y_j) \tag{5.8}$$

[2] En el código, estos 10^{11} nodos se traducen en 10^{11} iteraciones del bucle.

siendo x_i e y_i una serie de números distribuidos uniformemente en el rango $a \leq x \leq m$ y $b \leq x \leq n$ de forma que:

$$x_i = \frac{(2 \cdot m - 2 \cdot i + 1) \cdot a \ + \ (2 \cdot i - 1) \cdot b}{2 \cdot m} \tag{5.9}$$

$$y_j = \frac{(2 \cdot n - 2 \cdot j + 1) \cdot a \ + \ (2 \cdot j - 1) \cdot b}{2 \cdot n} \tag{5.10}$$

5.1.1.5. Demostración de la Transformada Rápida de Fourier

En este experimento se paraleliza un programa que demuestra la Transformada Rápida de Fourier (FFT, *Fast Fourier Transform*), ampliamente empleada en procesamiento de señales, resolución de ecuaciones en derivadas parciales o algoritmos de multiplicación rápida de grandes enteros.

En este caso, se utiliza una base binaria compleja ($n = 2^m$) y un método de autoclasificación [208]. Las distintas ejecuciones han sido realizadas estableciendo un total de 27 iteraciones del método, desde $n = 1$ hasta ($\log_2 n = 27$), de modo que en cada iteración se duplica el índice n hasta alcanzar el máximo ($n = 134.217.728$). El código fuente secuencial [209] presenta varios bucles que pueden ser paralelizados, sin embargo, los experimentos se centran en el que se encarga de realizar cada uno de los pasos unitarios necesarios (ver Código B.5, -pág. 137-). Las instrucciones de este bucle, que itera sobre el vector de elementos de los factores de rotación (*twiddle factors*)[3], es distribuido entre múltiples hilos de ejecución, dado que los vectores internos son independientes entre sí para cada iteración de bucle.

5.1.1.6. Factorización LU

Este programa implementa un algoritmo de descomposición LU (*Lower-Upper*) para la factorización de matrices, orientado a la resolución de sistemas lineales densos y frecuentemente empleado en el renderizado de gráficos. De este modo, dado el siguiente sistema algebraico:

$$Ax = b \tag{5.11}$$

donde A es una matriz $n \times n$ no singular con los elementos de sus diagonales distintos de cero, $b = (b_0, b_1, ..., b_{n-1})^T$ es el lado derecho de la ecuación y $x = (x_0, x_1, ..., x_{n-1})^T$ es el vector de incógnitas, la factorización LU transforma la matriz A en dos matrices L y U que cumplen:

$$A = LU \tag{5.12}$$

siendo L y U dos matrices triangulares inferior y superior respectivamente. De esta forma, la solución a la ecuación original 5.1.1.6 puede encontrarse resolviendo los siguientes sistemas triangulares:

$$Ly = b \tag{5.13}$$

[3] Coeficientes trigonométricos que son multiplicados por los datos originales durante la transformación.

$$Ux = y \tag{5.14}$$

La implementación del algoritmo emplea tres bucles anidados de manera que para cada iteración del bucle más externo, existe un paso de división y otro de eliminación [210], paralelizando el bucle de segundo nivel para procesar las columnas de forma concurrente. De este modo, el sistema es transformado en matrices elementales que representan los distintos pasos para su resolución (ver Código B.6, -pág. 137-).

5.1.1.7. Simulaciones de dinámica molecular

Este experimento está basado en simulaciones de dinámica molecular que emplean el algoritmo de Verlet [211] para la integración numérica de ecuaciones diferenciales ordinarias de segundo orden con valores iniciales conocidos. Las tareas computacionales permiten seguir los caminos de las distintas partículas, las cuales ejercen fuerzas entre sí en función de la distancia a la que se encuentren. El problema es tratado como un conjunto acoplado de ecuaciones diferenciales cuyo sistema es discretizado. En función de la velocidad y la posición de cada partícula en un instante de tiempo, el algoritmo estima ambas en el instante siguiente [212]. El Código B.7 (pág. 138) muestra la paralelización automática realizada sobre el bucle que calcula las fuerzas y energías de las partículas. En él, el potencial es calculado mediante la ecuación:

$$V(x) = \left(sin\left(min\left(x, \frac{\pi}{2}\right)\right)\right)^2 \tag{5.15}$$

y su derivada:

$$dv(x) = 2{,}0 \cdot sin\left(min\left(x, \frac{\pi}{2}\right)\right) \cdot cos\left(min\left(x, \frac{\pi}{2}\right)\right) \tag{5.16}$$

$$dv(x) = sin\left(2{,}0 \cdot min\left(x, \frac{\pi}{2}\right)\right) \tag{5.17}$$

5.1.1.8. Multiplicación de matrices densas

Experimento basado en la multiplicación de matrices densas donde se consideran tres matrices que cumplen la siguiente ecuación:

$$A = B * C \tag{5.18}$$

siendo B y C escogidas de forma que

$$A = (N+1) \cdot I \tag{5.19}$$

donde N es el orden de B y de C, e I es la matriz identidad.

En el Código B.8 (pág. 139) puede observarse el bucle paralelizado que efectúa la multiplicación. Para la ejecución del experimento se establece que B y C son matrices de orden 10^4.

5.1.1.9. Aproximación de la ecuación de Poisson

En este experimento se realiza una aproximación de la ecuación de Poisson en una región rectangular mediante el método iterativo de Jacobi. Se trata de una ecuación en derivadas parciales (PDE, *Partial Differential Equation*) ampliamente utilizada en electrostática, ingeniería mecánica y especialmente en física teórica. Este programa emplea en concreto la siguiente versión de la ecuación [213]:

$$\left(\frac{d}{dx}\frac{d}{dx}+\frac{d}{dy}\frac{d}{dy}\right)U(x,y)=F(x,y) \tag{5.20}$$

sobre un rectángulo $0 \leq x \leq 1$, $0 \leq y \leq 1$, con solución exacta

$$U(x,y)=sen(\pi \cdot x \cdot y) \tag{5.21}$$

utilizando $NX \times NX$ puntos espaciados uniformemente, y condiciones de frontera de Dirichlet [214] a lo largo de las líneas $x=0$, $x=1$, $y=0$, $y=1$. De este modo, la solución es calculada discretizando la geometría, asumiendo que $dx=dy$, así como aproximando el operador de Poisson mediante la siguiente ecuación:

$$(U(i-1,j)+U(i+1,j)+U(i,j+1)-4\times U(i,j)) \;/\; \frac{dx}{dy} \tag{5.22}$$

El transcompilador permite en este caso paralelizar correctamente el método iterativo de Jacobi que resuelve el sistema lineal de U (ver Código B.9, -pág. 139-).

5.1.2. Resultados

En este apartado se presentan los resultados obtenidos por el transcompilador respecto a la paralelización de códigos (apartado 5.1.2.1) y la predicción de la planificación OpenMP más adecuada, mediante la aplicación de aprendizaje automático supervisado (5.1.2.2).

5.1.2.1. Paralelización de códigos secuenciales

A continuación se presentan los resultados obtenidos en los distintos experimentos y se muestra cómo el transcompilador permite alcanzar significativas reducciones en los tiempos de ejecución (hasta un 97,65 % respecto a las versiones secuenciales originales), sin modificar manualmente ninguna línea de los programas y con grados de mejora que dependen de la propia estructura del código.

En primer lugar, el programa secuencial de cada experimento es paralelizado de forma automática mediante el transcompilador. A continuación, se utiliza el sistema de apoyo a la toma de decisiones (descrito en el apartado 3.2.4.1 del Capítulo 3) para generar también automáticamente distintas versiones de cada código paralelo, con el objetivo de identificar la combinación óptima con la que se alcanzan los mejores resultados. Las ejecuciones consideran los tres tipos de planificación paralela (estática, dinámica y

guiada), así como diferente número de cores (desde 1 hasta el máximo disponible por nodo). De esta forma, para cada experimento se han realizado 145 ejecuciones en los procesadores E7-4830v3 (3 planificaciones paralelas × 48 cores + la ejecución secuencial del código) y 61 ejecuciones en los procesadores E5-2660v3 (3 planificaciones × 20 cores + el procesamiento secuencial). En concreto, para cada experimento, el DSS ha modificado, compilado, enviado a las colas de ejecución y extraído automáticamente los resultados obtenidos. Como ejemplo, en el Código 5.1 se muestra (resumida por razones de legibilidad) la salida del DSS correspondiente al conjunto de Mandelbrot en el nodo del clúster. Asimismo, en la Figura 5.1 puede verse el código paralelo resultante en la interfaz gráfica del compilador.

La información correspondiente a las ejecuciones de los programas paralelos generados es expuesta en la Tabla 5.2, donde la velocidad y la eficiencia obtenidas son utilizadas como evaluación cuantitativa del código paralelizado. Los tiempos de ejecución se muestran en segundos (redondeados a dos decimales por cuestiones de legibilidad).

Código 5.1: Resultados (resumidos) del *Conjunto de Mandelbrot* ofrecidos por el DSS.

```
1
2  Analizando Mandelbrot...
3
4  Programa     Plan.     Cores    Tiempo     Tiempo(s)   Speedup   Eficiencia
5  ---------------------------------------------------------------------------
6  mandelbrot   estática     1    15:13.62     913,62     0,0000     0,0000
7  mandelbrot   estática     2     7:42.12     462,12     1,9770     0,9885   <--
8  mandelbrot   estática     3    14:01.59     841,59     1,0855     0,3618
9     ...          ...      ..      ...        ...        ...        ...
10 mandelbrot   estática    18     3:13.66     193,66     4,7176     0,2620
11 mandelbrot   estática    19     3:13.96     193,96     4,7103     0,2479
12 mandelbrot   estática    20     2:57.30     177,30     5,1529     0,2576
13 ---------------------------------------------------------------------------
14 mandelbrot   dinámica     1    15:14.02     914,02     0,9995     0,9995   <--
15 mandelbrot   dinámica     2     7:37.14     457,14     1,9985     0,9992   <--
16 mandelbrot   dinámica     3     5:07.60     307,60     2,9701     0,9900   <--
17    ...          ...      ..      ...        ...        ...        ...
18 mandelbrot   dinámica    18     0:57.96      57,96    15,7629     0,8757   <--
19 mandelbrot   dinámica    19     0:54.94      54,94    16,6294     0,8752   <--
20 mandelbrot   dinámica    20     0:52.20      52,20    17,5022     0,8751   <--
21 ---------------------------------------------------------------------------
22 mandelbrot   guiada       1    15:13.96     913,96     0,9996     0,9996   <--
23 mandelbrot   guiada       2     7:42.10     462,10     1,9771     0,9885   <--
24 mandelbrot   guiada       3    10:07.29     607,29     1,5044     0,5014   <--
25    ...          ...      ..      ...        ...        ...        ...
26 mandelbrot   guiada      18     1:53.39     113,39     8,0573     0,4476
27 mandelbrot   guiada      19     1:48.42     108,42     8,4266     0,4435
28 mandelbrot   guiada      20     1:40.10     100,10     9,1270     0,4563
29 ---------------------------------------------------------------------------
30
31 Mejores resultados (por planificación)...
32
33 Planificación  Cores   Tiempo     Speedup
34 ---------------------------------------------
35 secuencial       1    15:13.62     0,0000
36 ---------------------------------------------
37 estática        20     2:57.30     5,1529
38 dinámica        20     0:52.20    17,5022
39 guiada          20     1:40.10     9,1270
```

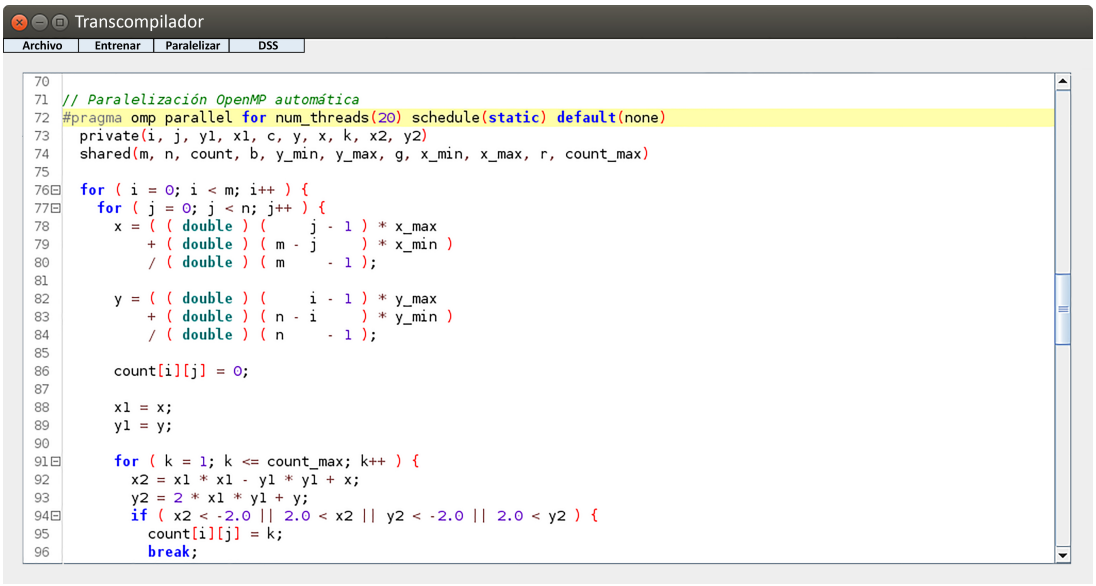

Figura 5.1: Intefaz gráfica del transcompilador. *Conjunto de Mandelbrot*.

Tabla 5.2: Resultados experimentales de paralelización automática considerando todas las planificaciones.

Experimento	Planificación	Intel Xeon E7-4830v3				Intel Xeon E5-2660v3			
		Cores	Tiempo (s)	Speedup	Eficiencia	Cores	Tiempo (s)	Speedup	Eficiencia
1 Conjunto de Mandelbrot	Secuencial	1	1.117,34	0,00	0,00	1	913,62	0,00	0,00
	Estática	48	99,80	11,20	0,23	20	177,30	5,15	0,26
	Dinámica	48	26,64	41,94	0,87	20	52,20	17,50	0,88
	Guiada	48	55,24	20,23	0,42	20	100,10	9,13	0,46
2 Números Primos	Secuencial	1	4.277,00	0,00	0,00	1	3.495,13	0,00	0,00
	Estática	48	142,45	30,02	0,63	20	278,10	12,57	0,63
	Dinámica	48	117,17	36,50	0,76	20	202,88	17,23	0,86
	Guiada	48	100,39	42,60	0,89	20	199,40	17,53	0,88
3 Integral $\int_a^b f(x)\,\mathrm{d}x$	Secuencial	1	3.852,00	0,00	0,00	1	3.150,60	0,00	0,00
	Estática	48	90,55	42,54	0,89	20	179,37	17,56	0,88
	Dinámica	2	11.788,00	0,33	0,16	3	10.535,00	0,30	0,10
	Guiada	48	90,44	42,59	0,89	20	179,31	17,57	0,88
4 Integral $\int_R f(x,y)\,\mathrm{d}x\mathrm{d}y$	Secuencial	1	3.123,12	0,00	0,00	1	2.554,04	0,00	0,00
	Estática	48	73,45	42,52	0,89	20	145,44	17,56	0,88
	Dinámica	48	73,35	42,58	0,89	20	145,40	17,57	0,88
	Guiada	48	73,38	42,56	0,89	20	145,39	17,57	0,88
5 Transformada R. Fourier	Secuencial	1	328,01	0,00	0,00	1	257,58	0,00	0,00
	Estática	39	95,88	3,42	0,09	16	91,54	2,81	0,18
	Dinámica	45	206,66	1,59	0,04	15	171,83	1,50	0,10
	Guiada	26	98,83	3,32	0,13	9	94,12	2,74	0,30
6 Factorización LU	Secuencial	1	6.306,00	0,00	0,00	1	5.132,00	0,00	0,00
	Estática	46	178,40	35,35	0,77	20	316,00	16,22	0,81
	Dinámica	47	166,08	37,97	0,81	20	342,30	14,99	0,75
	Guiada	47	166,90	37,78	0,80	18	396,14	12,96	0,72
7 Dinámica molecular	Secuencial	1	4.708,00	0,00	0,00	1	3.846,00	0,00	0,00
	Estática	48	112,58	41,82	0,87	20	219,74	17,50	0,88
	Dinámica	48	224,50	20,97	0,44	20	347,72	11,06	0,55
	Guiada	48	124,72	37,75	0,79	20	226,71	16,96	0,85
8 Multiplicación de matrices	Secuencial	1	17.172,00	0,00	0,00	1	12.747,00	0,00	0,00
	Estática	48	475,82	36,09	0,75	20	717,18	17,77	0,89
	Dinámica	48	474,01	36,23	0,75	20	717,58	17,76	0,89
	Guiada	48	469,82	36,55	0,76	20	715,79	17,81	0,89
9 Aproximación de Poisson	Secuencial	1	1.884,37	0,00	0,00	1	1.415,00	0,00	0,00
	Estática	25	255,14	7,39	0,30	19	205,81	6,88	0,36
	Dinámica	46	461,00	4,08	0,09	20	456,58	3,10	0,15
	Guiada	48	342,45	5,50	0,11	19	273,07	5,18	0,27

Como se indica en el apartado 2.2.4 del Capítulo 2 dedicado a las medidas de rendimiento de los programas paralelos, el *speedup* (S_p) es calculado dividiendo el tiempo de ejecución paralelo entre el tiempo secuencial del experimento. Por otro lado, la eficiencia E es el resultado de dividir el *speedup* entre el número de cores utilizados, representando de este modo la relación del grado de aceleración conseguido frente al valor máximo posible, siendo $0 \leq E \leq 1$. Así, la eficiencia más baja ($E = 0$) se corresponde con el hecho de que todo el programa se ejecute de forma secuencial, mientras que la máxima ($E = 1$) se obtendría empleando todos los cores al 100 % durante el tiempo completo de ejecución.

Asimismo, la tabla muestra el mejor resultado obtenido con cada planificación (secuencial, estática, dinámica y guiada), es decir, el que requiere un menor tiempo de ejecución. No obstante, cabe destacar que en algunos casos, como en el de la *Integral doble* (experimento 4) o la *Multiplicación de matrices densas* (8), las diferencias entre las distintas planificaciones no son destacables. Además, en aquellos casos donde a partir de un número determinado de cores el tiempo de ejecución no mejora en al menos 1 segundo, el DSS escoge el resultado con mayor eficiencia, es decir, el que obtiene dicho tiempo con un menor número de cores.

En el nodo con los 20 cores E5-2660v3 los tiempos son menores en todas las ejecuciones secuenciales (debido principalmente a su mayor frecuencia de reloj), mientras que los 48 cores E7-4830v3 obtienen tiempos más reducidos en la mayoría de las planificaciones paralelas[4], salvo en aquellas que no escalan adecuadamente. Esto sucede, por ejemplo, utilizando la planificación dinámica con la integral $\int_a^b f(x)$ (3), debido al significativo aumento del *overhead* provocado porque los tiempos de ejecución de cada iteración son demasiado pequeños y las cargas de trabajo están balanceadas (no disminuyen durante el recorrido del bucle). No obstante, este tipo de casos tendrían que ser analizados en mayor profundidad con el objetivo de intentar encontrar un tamaño de fragmento (*chunk*) óptimo (ver apartado 2.3.3 sobre OpenMP en el Capítulo 2), que permitiera mejorar los resultados con estas planificaciones, puesto que emplear trozos demasiado grandes genera también desequilibrios en las cargas de trabajo.

La Tabla 5.3 resume el contenido de la Tabla 5.2, mostrando la mejor ejecución para cada experimento a fin de facilitar las comparaciones entre ambos nodos de cómputo. Adicionalmente, las Figuras 5.2 y 5.3 son proporcionadas para facilitar la comprensión de estos datos. La primera muestra en su eje horizontal los nombres que identifican a los distintos experimentos, mientras que en los ejes verticales se ofrecen los porcentajes de reducción alcanzados en los tiempos de ejecución paralelos (respecto a sus versiones secuenciales), comparando la utilización de los procesadores E7-4830v3 correspondientes al nodo de memoria compartida (barras amarillas) y los E5-2660v3 (barras rojas), pertenecientes al clúster de memoria distribuida. De forma similar, en la Figura 5.3 puede observarse la eficiencia alcanzada en cada uno de los experimentos paralelizados.

[4]Dado que los códigos son paralelizados mediante directivas OpenMP, sus ejecuciones no aprovechan las posibilidades de la distribución de tareas entre otros cores. Para mejorar los resultados en el clúster con los procesadores E5-2660v3 habría que combinar OpenMP con otros modelos de programación como, por ejemplo, MPI.

Tabla 5.3: Resultados experimentales de paralelización automática considerando la mejor planificación en cada caso.

	Experimento	Intel Xeon E7-4830v3					Intel Xeon E7-4830v3				
		Planificación	Cores	Tiempo	Speedup	Efic.	Planificación	Cores	Tiempo	Speedup	Efic.
1	**Conjunto de Mandelbrot**	Dinámica	48	26,64	41,94	0,87	Dinámica	20	52,20	17,50	0,88
2	**Números Primos**	Guiada	48	100,39	42,60	0,89	Dinámica $^{g\cdot}$	20	199,40	17,53	0,88
3	**Integral** $\mathrm{d}x$	Estática $^{g\cdot}$	48	90,44	42,59	0,89	Estática $^{g\cdot}$	20	179,31	17,57	0,88
4	**Integral doble** $\mathrm{d}x\mathrm{d}y$	*	48	73,35	42,58	0,89	*	20	145,39	17,57	0,88
5	**Transformada R. Fourier**	Estática $^{g\cdot}$	39	95,88	3,42	0,09	Estática $^{g\cdot}$	16	91,54	2,81	0,18
6	**Factorización LU**	Dinámica $^{g\cdot}$	47	166,08	37,97	0,81	Estática	20	316,00	16,22	0,81
7	**Dinámica molecular**	Estática	48	112,58	41,82	0,87	Estática	20	219,74	17,50	0,88
8	**Multiplicación matrices**	Guiada	48	469,82	36,55	0,76	*	20	717,18	17,77	0,89
9	**Aproximación de Poisson**	Estática	25	255,14	7,39	0,30	Estática	19	205,81	6,88	0,36

$^{g\cdot}$ Resultados similares en la planificación Guiada.
* Resultados similares en las tres planificaciones.

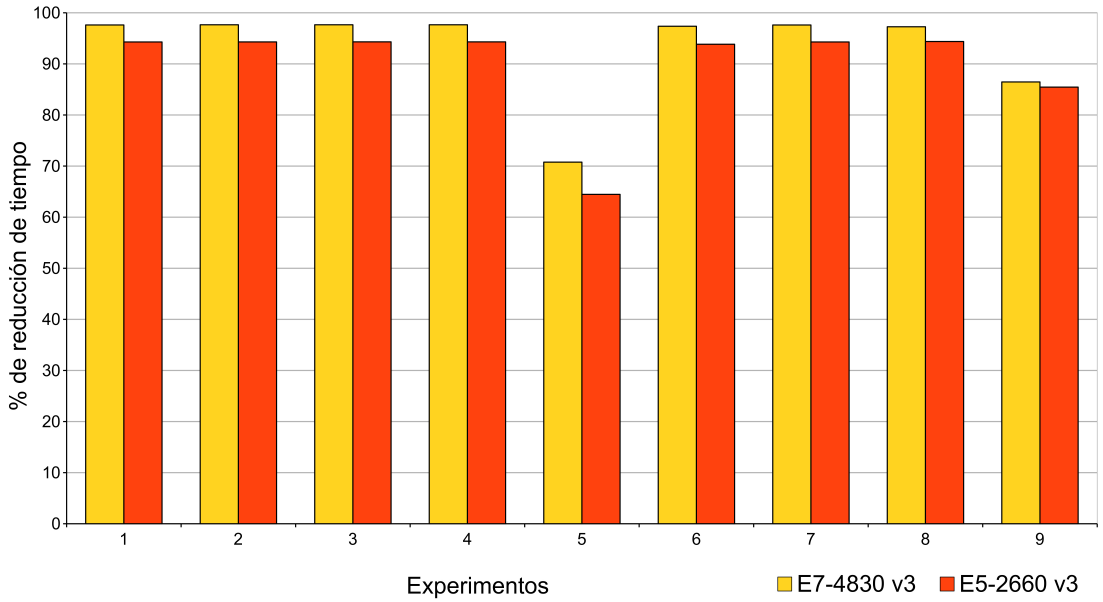

Figura 5.2: Porcentajes de reducción de tiempos de ejecución obtenidos mediante la paralelización automática realizada por el transcompilador.

La comparación entre los *speedups* alcanzados ha sido omitida al depender del número total de cores de ejecución (distinto en cada nodo de cómputo utilizado). No obstante, la eficiencia, que sí aparece representada, depende de este parámetro. En cualquier caso, en el nodo de memoria compartida los experimentos alcanzan una aceleración de 42,60 (frente a un máximo teórico de 48), con una aceleración superior a 41 en cinco de los experimentos. En el nodo de memoria distribuida se consiguen *speedups* de hasta 17,77 puntos (frente a un máximo teórico de 20), con siete experimentos con valores superiores a los 16 puntos.

Como se puede observar en las Figuras 5.2 y 5.3, los códigos paralelizados por el transcompilador obtienen reducciones entre el 64 y el 97 % respecto a los tiempos secuenciales originales, con eficiencias que llegan a alcanzar un índice de 0,89. Especialmente destacables son los resultados logrados con el *conjunto de Mandelbrot* (1), el *cálculo de números primos* (2), las *estimaciones de ambas integrales* (3 y 4), las *simulaciones de dinámica molecular* (7) y la *multiplicación de matrices densas* (8), que obtienen mejoras entre el 94,29 y el 97,65 % con eficiencias entorno a 0,88 en ambos nodos de cómputo.

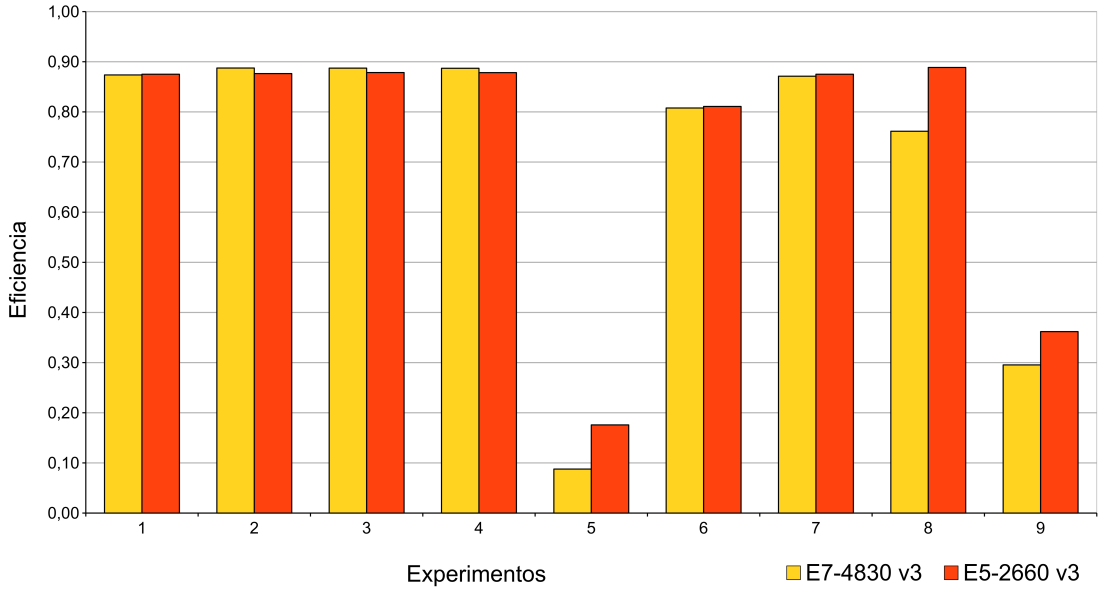

Figura 5.3: Eficiencia alcanzada en los experimentos mediante la paralelización automática realizada por el transcompilador.

En los experimentos relacionados con la *factorización LU* (6), la mejora se mantiene pero la eficiencia decrece hasta 0,81. Los peores resultados son obtenidos con la *aproximación de la ecuación de Poisson* (9), donde la reducción es del 86 % con eficiencias entre 0,30 y 0,30, y la *Transformada Rápida de Fourier* (5), que consigue tan solo una mejora alrededor del 70,77 % con una eficiencia de 0,09 en el nodo de memoria compartida (64,46 % y 0,18 en distribuida), debido principalmente al alto número de líneas de código secuenciales de ambos códigos.

En la Figura 5.3, las diferencias existentes entre ambos procesadores respecto a las eficiencias de la *Transformada Rápida de Fourier* (5), la *multiplicación de matrices densas* (8) y la *aproximación de la ecuación de Poisson* (9), son debidas al uso de distinto número de cores y a los desiguales tiempos de ejecución (como se ha indicado anteriormente, derivados de las características particulares de cada nodo).

En el caso de la *Transformada Rápida de Fourier* además se debe tener en cuenta que los cálculos realizados en los bucles internos realizan tareas computacionales de poca intensidad, por lo que no contrarrestan suficientemente el *overhead* asociado a la paralelización [208]. En general, el hecho de crear un gran número de tareas de grano muy fino implica la utilización de un significativo tiempo de creación y suspensión de los hilos de ejecución, de forma que sería necesario explotar de manera más eficiente la localidad de los datos para reducir en lo posible el *overhead* generado [161]. En este sentido, hay que recordar además que la mejora máxima siempre va a estar limitada, tanto por la proporción existente entre el código secuencial y el paralelizado, como por las condiciones en el reparto de las tareas en los bucles paralelos (debido a la naturaleza del propio código y al equilibrio de las cargas de trabajo). Por ello, esta tesis propone adicionalmente una serie de técnicas *software* destinadas a la escritura de códigos de programación eficientes, que permiten mejorar notablemente el rendimiento de ejecución de estas partes secuenciales.

Es importante subrayar también que la elección de una planificación inadecuada puede implicar incluso un destacable incremento de tiempo respecto a la ejecución secuencial, malgastando por tanto recursos de cómputo. Esto sucede, por ejemplo, en el caso de la integral $\int_a^b f(x)\,dx$ (3), donde el mejor resultado de la planificación dinámica es obtenido con dos únicos cores e implica una penalización del 206 % respecto al tiempo secuencial (ver Tabla 5.2). En este sentido, aunque la planificación estática presenta buenos resultados en la mayoría de los experimentos, tampoco puede ser considerada siempre como la más adecuada, puesto que en casos como el *conjunto de Mandelbrot* (1) y el *cálculo de números primos* (2), sus resultados son significativamente mejorados por las otras planificaciones.

Asimismo, la Figura 5.4 expone una representación gráfica de los resultados obtenidos en los procesadores E5-2660v3 con los experimentos de la *Transformada Rápida de Fourier* (5). El eje X muestra los tres tipos de planificación y el eje Y izquierdo ofrece el porcentaje de reducción del tiempo de ejecución (barras rojas), mientras que en el eje Y derecho se reproducen las distintas eficiencias (cuadrados negros unidos por una línea negra con objeto de facilitar su ubicación). Puede observarse cómo con una planificación guiada con 9 cores se obtiene prácticamente el mismo tiempo de ejecución que utilizando 16 cores con planificación estática (reducciones de tiempo del 63,46 % y el 64,46 respectivamente). De este modo, al elegir la planificación estática se infrautilizan varios cores (eficiencia de 0,30 en guiada, frente a 0,18 en estática).

En consecuencia, este experimento demuestra la importancia de elegir una correcta planificación y motiva asimismo una nueva línea de trabajo, que pretende afrontar la predición del número de cores óptimos para alcanzar un determinado *speedup* sin afectar significativamente a la eficiencia, es decir, persiguiendo no desaprovechar los recursos de cómputo empleados. En este sentido, se detallan a continuación las pruebas realizadas respecto al optimizador de código del transcompilador.

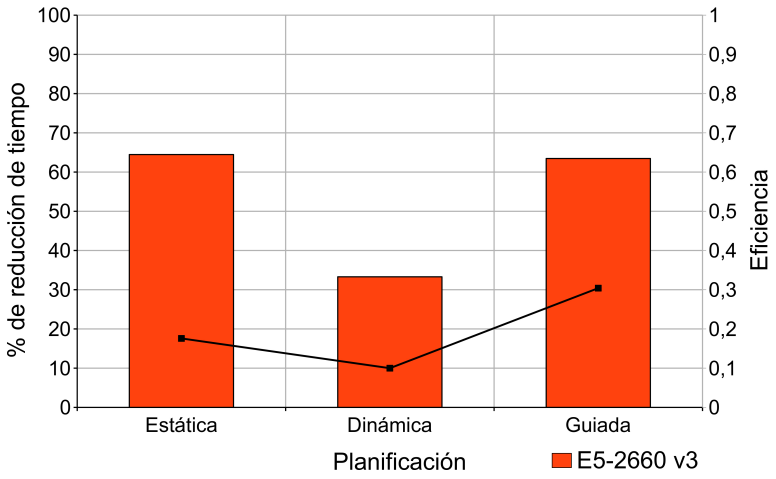

Figura 5.4: Porcentaje de reducción del tiempo de ejecución y eficiencia de la Transformada Rápida de Fourier en los procesadores Intel Xeon E5-2660v3 según la planificación paralela utilizada.

5.1.2.2. Predicción de la planificación OpenMP

En este apartado se analizan los resultados obtenidos para demostrar la funcionalidad del optimizador, publicados parcialmente en [202]. Para ello se utilizan cinco de los experimentos referenciados en el apartado anterior: *conjunto de Mandelbrot* (experimento 1), *cálculo de números primos* (2), *estimaciones de integrales* (3 y 4) y *multiplicación de matrices densas* (8). Estos experimentos han sido escogidos frente al resto en virtud de las particularidades de sus bucles, puesto que debido a la naturaleza del paralelizador, las características utilizadas por el optimizador (ver apartado 3.2.4.2) deben ser estáticas, es decir, conocidas en tiempo de compilación. De este modo, se consideran siempre bucles con límites inferiores y superiores constantes, así como iteradores de control también invariantes[5].

El objetivo ha consistido en comprobar si el optimizador predice correctamente la estrategia de planificación OpenMP más adecuada para realizar las ejecuciones. Así, los experimentos han sido paralelizados de forma automática y clasificados en la planificación (estática, dinámica o guiada) que el optimizador predice como más recomendable. Para ello se considera siempre el número de cores totales del nodo de cómputo, aunque se prevé extender la funcionalidad del clasificador de forma que considere también la eficiencia y el número de cores más adecuado para cada ejecución.

El algoritmo de los *k*-vecinos más cercanos (descrito en el apartado 3.2.4.2) ha demostrado su eficacia ante un número reducido de problemas. Sin embargo, presenta dos inconvenientes: en primer lugar, debe calcular en cada predicción la distancia entre las características del bucle a optimizar y las relativas a todos los bucles del conjunto de entrenamiento, lo cual puede llegar a implicar una significativa cantidad de tiempo a medida que se incrementa el número de ejemplos del conjunto. Además, este algoritmo no aprende realmente de los datos de entrenamiento, sino que directamente selecciona los vecinos más cercanos, lo cual no garantiza la robustez del mismo frente a ejemplos que, estando correctamente clasificados (al haber sido previamente ejecutados y etiquetados con la planificación correcta) presenten valores atípicos que posteriormente infieran predicciones erróneas. Por tanto, será necesario explorar técnicas de clasificación adicionales como árboles de decisión (*decision tree*) o bosques aleatorios (*random forests*) [215], así como métodos de validación cruzada como *leave one out* [216] o *k-fold* [217]. Asimismo, el optimizador asume que las cargas de trabajo no varían durante su ejecución. Para analizar cambios dinámicos en las mismas sería adecuado emplear otros algoritmos basados en ajuste automático.

De este modo, en la implementación del aprendizaje del optimizador se han tenido en cuenta las siguientes consideraciones: la planificación estática es más adecuada para bucles únicos (sin ningún nivel de anidación adicional) con iteraciones distribuidas de forma uniforme y sin desequilibrio o bien con cargas de trabajo relativamente pequeñas [148]. Sin embargo, esta planificación no es conveniente cuando por alguna razón la cantidad de trabajo por iteración no permanece constante, sino que por ejemplo, disminuye con el índice del bucle, originando desequilibrios de carga. En estos casos, presentan mejores resultados las planificaciones dinámica y guiada. En la primera, las iteraciones van siendo

[5] Con un comportamiento completamente predefinido en tiempo de compilación.

asignadas a los hilos de ejecución a medida que estos finalizan su fragmento actual. Esto reduce el desequilibrio, puesto que aunque los hilos con cargas de trabajo menores terminen antes, siguen ejecutando otras iteraciones. No obstante, la planificación dinámica genera excesivo *overhead* cuando los fragmentos son demasiado pequeños en términos de tiempo de ejecución [63] (como sucede al aplicar esta estrategia en el experimento de la integral $\int f(x)\,\mathrm{d}x$ descrito anteriormente. Ver Tabla 5.2, -pág. 69-). De forma similar, utilizar fragmentos de mayor tamaño con esta planificación también aumenta el desequilibrio. Por tanto, cuando se cumplen estas condiciones, el bucle se clasifica en la estrategia guiada, similar a la dinámica, pero en la cual el tamaño del fragmento va disminuyendo en tiempo de ejecución y es siempre proporcional a la relación entre el número restante de iteraciones del bucle (las que aún no han sido asignadas) y el número de hilos de ejecución [184]. Así, al comenzar la ejecución se procesan los trozos más grandes y a medida que ésta avanza, el sistema utiliza fragmentos más pequeños para llenar los huecos en la planificación, consiguiendo en estos casos un mejor rendimiento y aprovechamiento de los recursos.

La Figura 5.5 representa gráficamente los resultados obtenidos al emplear el algoritmo de aprendizaje con los cinco experimentos paralelizados automáticamente por el transcompilador. En el eje de abscisas se muestran los números que los identifican, mientras que en el de ordenadas se ofrecen los porcentajes de reducción de tiempo obtenidos en cada experimento (mostrando únicamente una escala entre el 95 % y el 100 % por razones de legibilidad). La gráfica permite comparar los mejores resultados obtenidos en el procesador E7-4830v3 (barras amarillas) con los alcanzados mediante las ejecuciones predichas.

El transcompilador realiza la predicción óptima en el *conjunto de Mandelbrot* (1), determinando una planificación dinámica, la *estimación de la integral* $\int f(x)\,\mathrm{d}x$ (3), escogiendo la estática, y la *multiplicación de matrices densas* (8), usando la guiada. Por contra, en el resto de experimentos el algoritmo no acierta en la predicción, aunque el impacto en los resultados no es significativo. En el caso del *cálculo de números primos* (2), se establece la estrategia estática, que emplea 42,06 segundos más de ejecución respecto

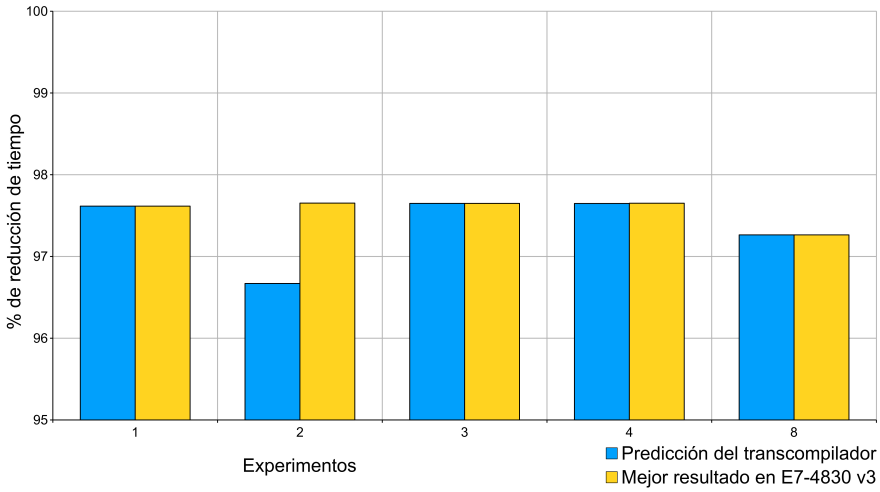

Figura 5.5: Porcentajes de reducción del tiempo de ejecución en el procesador E7-4830v3. Comparación de los mejores resultados con los obtenidos al utilizar la planificación predicha de forma automática por el transcompilador.

a la planificación guiada, lo cual se traduce en prácticamente un punto porcentual menos de reducción (96,67 % frente a 97,65 %). De forma similar, en la *estimación de la integral doble* $\int f(x,y)\,\mathrm{d}x\mathrm{d}y$ (4) el algoritmo también predice la planificación estática en lugar de la dinámica, aunque en este caso la diferencia es despreciable, con una diferencia de 0,1 segundos y 0,003 %.

Aunque las pruebas desarrolladas han mostrado unos resultados apropiados, tal y como sucede en el resto de propuestas similares existentes en la literatura (ver apartado 2.3.4 sobre la aplicación de *machine learning* a la paralelización OpenMP), estos no pueden ser generalizados más allá de los propios programas empleados, las cargas de trabajo configuradas y los dominios de aplicación utilizados [149]. Además, los resultados obtenidos no pueden ser considerados especialmente relevantes dado el reducido número de experimentos desarrollados. En cualquier caso, el objetivo primordial de estos experimentos ha sido asegurar la correcta funcionalidad del optimizador de cara a próximas investigaciones, centradas en ampliar el alcance del mismo.

Además, es importante considerar que ninguna planificación puede abordar todos los aspectos que influyen en el desequilibrio de las cargas, especialmente en relación a las características de los sistemas de cómputo utilizados, que también pueden determinar, generalmente en tiempo de ejecución, el rendimiento de cada estrategia [218]. De hecho, algunos estudios demuestran que las planificaciones OpenMP disponibles son insuficientes para cubrir todas las combinaciones de aplicaciones, sistemas y variabilidad de sus características [218, 219] y se han propuesto técnicas adicionales para la planificación de bucles [170, 220, 221]. En cualquier caso, los resultados obtenidos serán ampliados en trabajos futuros utilizando un número ampliamente mayor de experimentos.

Con objeto de mejorar el algoritmo de aprendizaje, se ha desarrollado también un método piloto de medición de *overheads* (publicado en [222]), que presenta como objetivo poder analizar los efectos producidos, tanto por el incremento del número de cores de ejecución, como por la planificación paralela escogida en cada caso, demostrando su efectividad en las pruebas preliminares realizadas sobre el cálculo de *overheads* asociados a bucles y operaciones paralelas representativas, estimando su influencia en los tiempos de ejecución. Se prevé asimismo perfeccionar este método y considerar además la utilización de modelos rendimiento [223], que permitan predecir con exactitud el tiempo empleado por las comunicaciones, ayudando así a mejorar el diseño y la implementación de los algoritmos.

5.2. Aplicación de las técnicas para el desarrollo de códigos eficientes

Adicionalmente a la paralelización automática de códigos, con el objetivo de complementar las ventajas ofrecidas por el transcompilador y dadas las limitaciones actuales del mismo (ver apartado 6.2 sobre limitaciones y trabajos futuros), en este capítulo se analizan y evalúan los resultados experimentales obtenidos respecto a la aplicación de las técnicas propuestas en el Capítulo 4, destinadas a mejorar la eficiencia de los códigos de programación.

Los experimentos han sido desarrollados en primer lugar sobre dispositivos IoT, debido a que en ellos las pruebas y mediciones a acometer presentan una menor complejidad, especialmente en relación al impacto de las técnicas sobre el consumo energético. De este modo, se ha concretado previa y adecuadamente el alcance de los experimentos a implementar para cada una de las técnicas, consiguiendo evitar la utilización para ello de un número excesivo de horas de cómputo en infraestructuras HPC. Además, los dispositivos IoT utilizados cuentan con arquitecturas ARM, permitiendo por tanto que los resultados puedan ser extrapolados a un mayor número de sistemas de cómputo. Posteriormente, una vez demostrada su eficiencia en la reducción de los tiempos de ejecución y en la mejora del consumo energético, estas mismas estrategias han sido aplicadas sobre códigos de programación en infraestructuras de cómputo de alto rendimiento. De esta forma se ha comprobado que las técnicas propuestas permiten mejorar la eficiencia del código de forma relativamente fácil y sencilla, posibilitando que los usuarios obtengan mejoras significativas con cambios menores en sus códigos.

A continuación se describen las distintas secciones que conforman este apartado: en el punto 5.2.1 se analiza detalladamente el análisis del impacto producido al emplear estas técnicas en dispositivos del Internet de las cosas, evaluando sus resultados respecto a la reducción de los tiempos de ejecución (apartado 5.2.1.1) y el ahorro en el consumo de energía (5.2.1.2). Por último, en el punto 5.2.2 se consideran los resultados obtenidos al aplicar las estrategias en infraestructuras HPC.

5.2.1. Aplicación de las técnicas en dispositivos IoT

El Internet de las cosas, que tiene como objetivo mejorar la calidad de vida mediante la conexión de dispositivos, aplicaciones y tecnologías inteligentes, ha crecido exponencialmente en los últimos años potenciado principalmente por el gran volumen de nuevos sensores, protocolos y comunicaciones disponibles. De este modo, el IoT permite que todo tipo de objetos físicos se conecten a Internet e interactúen sin necesidad de intervención humana, convirtiendo estos elementos en dispositivos inteligentes [224]. Sin embargo, a pesar de su potencial actual, aún debe enfrentarse a múltiples desafíos, principalmente relacionados con el direccionamiento y el enrutamiento, la estandarización, la seguridad y la privacidad, o las limitaciones físicas propias de este tipo de dispositivos, por ejemplo, aquellas relativas al consumo energético, la capacidad de procesamiento o la memoria máxima disponible [225].

Los ordenadores de placa simple (*Single-board computers*, SBC), también denominados de placa única o reducida, están siendo ampliamente utilizados en el desarrollo de sistemas IoT. Aunque su rendimiento no puede competir con la capacidad de cómputo de dispositivos de mayor precio y prestaciones, a menudo suponen la mejor alternativa para construir sistemas económicos, móviles y de bajo consumo, con un diseño compacto y una óptima relación coste/rendimiento. De esta forma, su uso en proyectos de investigación está ganando cada vez más interés.

Tras analizar distintas alternativas, se ha decidido emplear los dispositivos Raspberry Pi [226] (en adelante RPi), desarrollados por la Raspberry Pi Foundation [226], por diversas razones: en primer lugar, permiten computar datos en tiempos razonables en relación a la complejidad del problema[6] y de esta forma, al soportar una carga computacional mayor, son más adecuados para demostrar el impacto producido por las técnicas propuestas; además, la potencia computacional ofrecida por vatio permite realizar tareas esenciales de cálculo de forma más eficiente, consumiendo menos potencia y ahorrando una cantidad notable de energía; por último, su arquitectura ARM predomina ampliamente en IoT[7], por lo que los resultados obtenidos son extrapolables a un importante número de sistemas semejantes.

Además, presentan una mejor relación entre coste y rendimiento que otros dispositivos, encontrándose actualmente entre los SBC más utilizados, junto con los desarrollados por Arduino [227] (en febrero de 2020, la Fundación RPi anunció que se habían vendido 30 millones de sus dispositivos desde 2012). De hecho, están siendo ampliamente utilizados en gran variedad de proyectos de investigación representativos y heterogéneos que incluyen, por ejemplo redes de sensores [228–230], sistemas de seguridad activa en vehículos [231], asistencia sanitaria y telemedicina [232], analítica de big data [233] y análisis de imágenes [234], entre otras muchas áreas. También han sido ampliamente empleadas en proyectos educativos [235] y en la construcción de clústeres económicos y energéticamente eficientes con gran número de nodos [236] que compensan parcialmente su baja potencia computacional.

Los experimentos desarrollados han tenido como objetivo analizar la aplicación de las técnicas propuestas para la escritura eficiente de programas (detalladas en el Capítulo 4) en dispositivos RPi, preservando la semántica del programa. Para ello se evalúan los resultados obtenidos mediante la aplicación manual de estas técnicas, en comparación con los correspondientes a la utilización de las opciones de optimización ofrecidas por el compilador GCC [13]. Los experimentos demuestran que las técnicas logran importantes reducciones en los tiempos de ejecución y en el consumo de energía, incluso mejorando el impacto producido por las optimizaciones automáticas ofrecidas por el compilador. Así, se pone de manifiesto que estas últimas no alcanzan las mismas mejoras que los programadores pueden lograr mediante la aplicación de las técnicas propuestas.

[6] A diferencia de otros dispositivos IoT, las RPis no requieren que sus datos sean enviados a otros sistemas para poder ser procesados.

[7] En particular, el uso de procesadores ARM en este tipo de dispositivos ha crecido exponencialmente en los últimos años debido precisamente a su bajo consumo energético.

De este modo, se persigue ayudar a los programadores a escribir directamente código eficiente y demostrar que la ayuda del compilador es en algunos casos insuficiente, cuando se persigue que los programas sean lo más eficientes posibles.

5.2.1.1. Reducción del tiempo de ejecución en IoT

La primera parte de los experimentos desarrollados sobre dispositivos IoT ha sido centrada exclusivamente en la reducción de los tiempos de ejecución, sin tener en cuenta el consumo energético. Los resultados de estos experimentos fueron publicados sin incluir el modelo RPi de cuarta generación [237] al haber empezado a comercializarse posteriormente a la realización de estos estudios. Sin embargo, para la redacción de este trabajo, los experimentos han sido ampliados incluyendo este modelo (RPi 4B), que sí fue considerado ya en el siguiente artículo dedicado al impacto de las técnicas sobre el ahorro energético [238]. Lo mismo sucede con la técnica *conjuntos de bits* (T2), cuyo análisis solo es recogido en las publicaciones referentes a la aplicación de las técnicas en infraestructuras HPC [239, 240] y cuyos experimentos sobre IoT han sido también desarrollados con posterioridad con objeto de ser incluidos en esta tesis.

A continuación se detallan la metodología utilizada y los resultados alcanzados en la investigación realizada sobre la mejora de los tiempos de ejecución en estos dispositivos del Internet de las cosas [237].

5.2.1.1.1. Metodología

Los experimentos han sido desarrollados utilizando cuatro tipos de dispositivos Raspberry Pi: modelo B de segunda generación, modelos B y B+ de tercera generación y modelo B de cuarta generación, cuyas características se muestran en la Tabla 5.4 [226].

Las placas RPi pueden ser alimentadas de múltiples formas, entre las que destacan utilizar su puerto (micro-USB o USB-C) de 5V y suministrar la energía mediante un

Tabla 5.4: Especificaciones de los modelos Raspberry Pi (RPi) 2B, 3B, 3B+ y 4B.

Modelo	RPi 2 B	RPi 3 B	RPi 3 B+	RPi 4 B
SoC	Broadcom BCM2836	Broadcom BCM2837	Broadcom BCM2837B0	Broadcom BCM2711
CPU	4 x Arm Cortex-A7, 900 MHz	4 x Arm Cortex-A53, 1,2 GHz	4 x Arm Cortex-A53, 1,4 GHz	4 x Arm Cortex-A72, 1,5 GHz
RAM	1 GB	1 GB	1 GB	1 GB/2 GB/4 GB
GPU	Broadcom VideoCore IV	Broadcom VideoCore IV	Broadcom VideoCore IV	Broadcom VideoCore VI
Puertos USB	4	4	4	4 (2 x USB 3.0 + 2 x USB 2.0)
Ethernet	100 Mbit/s base Ethernet	100 Mbit/s base Ethernet	Gigabit Ethernet (max. 300 Mbps)	Gigabit Ethernet (no limit)
Power over Eth.	No	No	Yes (requires separate PoE HAT)	Yes (requires separate PoE HAT)
WiFi	No	WiFi 802.11n	WiFi 802.11ac Dual Band	WiFi 802.11ac Dual Band
Bluetooth	No	4.1	4.2 BLE	5.0 BLE
Salida video	HDMI/3,5 mm Comp./DSI	HDMI/3,5 mm Comp./DSI	HDMI/3,5 mm Comp./DSI	micro-HDMI/3,5 mm Comp./DSI
Salida audio	I^2 S/HDMI/3,5 mm Composite	I^2 S/HDMI/3,5 mm Composite	I^2 S/HDMI/3,5 mm Composite	I^2 S/HDMI/3,5 mm Composite
Entrada cámara	15 Pin CSI	15 Pin CSI	15 Pin CSI	15 Pin CSI
Pins GPIO	40	40	40	40
Memoria	MicroSD	MicroSD	MicroSD	MicroSD

adaptador de corriente o una batería portátil. También es posible emplear el puerto Ethernet, pero esta opción no se proporciona de manera estándar (como se indica en la tabla). Además, es posible utilizar el puerto de entrada/salida de propósito general (GPIO, *General Purpose Input/Output*) de cada Raspberry y conectar los pines de 5V a una fuente de energía que alimente directamente la placa.

Para analizar el impacto de la aplicación de las técnicas *software* sobre los tiempos de ejecución, los dispositivos han sido alimentados mediante la opción del adaptador de corriente, por su estabilidad y fiabilidad a la hora de suministrar energía. Todos los experimentos se han desarrollado a una temperatura ambiente de 22 °C, registrando las RPi una temperatura máxima de CPU de 60 °C durante las pruebas. Además, la programación realizada cuenta con pausas automáticas de 3 minutos entre las ejecuciones de cada experimento, con objeto de evitar posibles sobrecalentamientos durante las mediciones que puedan alterar los resultados.

Tal y como recomienda la Fundación Raspberry Pi [226], se utiliza el sistema operativo oficial del dispositivo (anteriormente llamado Raspbian y denominado Raspberry Pi OS en la actualidad), una distribución de GNU/Linux basada en Debian y optimizada para el conjunto de instrucciones ARM. En concreto, en estos experimentos se emplea la versión Raspbian Stretch 4.14 en cada uno de los dispositivos, con todos sus paquetes actualizados. Tanto el sistema operativo como los archivos relacionados con las pruebas realizadas, utilizan como almacenamiento tarjetas de memoria microSDHC clase 10 *SanDisk Ultra*, con una velocidad de escritura mínima de 10 MB/s.

Todos los códigos desarrollados han sido compilados con la versión 6.3.0 de GCC [13], contenida en la versión estable del paquete *Raspbian 6.3.0-18 + rpi1 + deb9u1* de la distribución. De este modo, para analizar las mejoras obtenidas mediante la optimización automática, se emplean los cuatro niveles de optimización ofrecidos por el compilador:

- Nivel 0: sin ninguna optimización sobre el tiempo de ejecución. Se trata del nivel predeterminado. Mientras en el resto se incrementa el tiempo de compilación, en este nivel el objetivo es reducir el coste de la compilación y posibilitar que en posibles depuraciones del código sigan obteniéndose siempre los resultados correctos. Es decir, si se detiene la ejecución del programa con un punto de interrupción, este nivel posibilita, por ejemplo, asignar un nuevo valor a cualquier variable y seguir obteniendo los resultados esperados con dicho cambio. En el resto de niveles de optimización el compilador intenta mejorar el rendimiento y el tamaño del código a expensas del tiempo de compilación, perdiendo en muchos casos la capacidad de depurar el programa.
- Nivel 1: optimización básica del tiempo de ejecución y el tamaño del código, sin aplicar optimizaciones que requieran mucho tiempo de compilación. En este nivel, la compilación presenta una duración algo superior, mientras que se emplea bastante más memoria en el caso de funciones grandes.
- Nivel 2: realiza una optimización mayor, aplicando la mayoría de las optimizaciones permitidas que no llegan a comprometer la velocidad de compilación o el espacio. En comparación con el nivel anterior, esta opción incrementa el rendimiento del código generado en detrimento del tiempo de compilación.

- Nivel 3: máximo nivel de optimización. Realiza todas las optimizaciones posibles buscando el mayor rendimiento del código a expensas del tiempo de compilación y, como se indicaba anteriormente, alterando la capacidad de depurar el programa.

Dependiendo tanto del objetivo como de la propia configuración de GCC, es posible habilitar un conjunto de optimizaciones diferentes en los niveles descritos. Todos los detalles sobre los indicadores de optimización pueden encontrarse en [13]. Del mismo modo, también es posible obtener el conjunto exacto de optimizaciones habilitadas en cada nivel mediante la invocación del comando "`gcc`" con los argumentos "`-Q -help = optimizers`".

Tanto *profilers* como herramientas de análisis permiten mejorar códigos de programación [38–40], pero habitualmente se centran en detectar cuellos de botella y medir los tiempos de ejecución de cada fragmento de código y no en mejorar la implementación del programa escribiendo directamente código eficiente, por lo que no son considerados dentro del alcance de esta investigación. Además, aunque los *profilers* son diseñados cuidadosamente, tienden a introducir latencias y *overheads* que pueden influir en el tiempo de ejecución, alterando las mediciones a realizar. Por ello, se ha decidido aplicar un método de medición más apropiado, descrito a continuación.

Se han desarrollado dos tests específicamente para cada técnica propuesta, con el objetivo de demostrar su eficiencia: un test con código estándar y un segundo test con código eficiente obtenido al aplicar la correspondiente técnica sobre el primero. El pseudocódigo general seguido en todos los experimentos es mostrado en el Algoritmo 6. Como se puede observar, se trata de un algoritmo de tiempo lineal con complejidad $O(n)$, siendo n el número de iteraciones del bucle. La complejidad interna de cada test varía dependiendo de cada técnica aunque, como se muestra en el código fuente de los distintos tests (ver Apéndice C), la complejidad es constante o lineal en la mayoría de los casos, excepto en los correspondientes a la técnica de *acceso por fila principal* (T5), debido al bucle doblemente anidado utilizado para recorrer la matriz bidimensional, que genera una complejidad cuadrática ($O(n^2)$) al tratarse de una matriz cuadrada.

Para llevar a cabo cada prueba asegurando la precisión de los resultados, cada test es repetido cien millones de veces para cada medición realizada (observar en la línea 5 el bucle con índice de iteración j). Esto permite que el tiempo de ejecución del código a analizar alcance un valor medible, posibilitando a su vez descartar el coste de tiempo asociado a la invocación de la función que implementa el test. Además, con objeto de reducir el impacto de los arranques en frío y los efectos de la memoria caché, se realizan diez mediciones (correspondientes al bucle con índice de iteración i -línea 3-). De esta manera, se cancelan las posibles variaciones aleatorias y el estado de la caché tiende a converger a un único valor, evitando así resultados atípicos (*outliers*). Tras calcular la media de esas diez mediciones, el resultado es dividido por el número de repeticiones (100.000.000), obteniendo así el tiempo consumido por el test en una única ejecución.

A su vez, con el objetivo de controlar las optimizaciones automáticas del compilador, las funciones que implementan cada test incluyen un atributo `OPTIMIZE` en su declaración que es definido mediante directivas de preprocesamiento (ver Código 5.2). Así, para compilar

Algoritmo 6: Medición de tiempo de ejecución.

```
1 Función obtenerTiemposEjecucionTest():
2     inicialización;
3     para i=0 hasta 9 hacer
4         iniciarTemporizador;
5         para j=0 hasta 100.000.000 hacer
6             ejecutarTest;
7         obtenerTiempo;
8     procesarEstadísticas;
9     devolver estadísticas;
```

Código 5.2: Directivas de preprocesamiento.

```
1 #ifdef level
2 #define OPTIMIZE __attribute__((optimize(level)))
3 #else
4 #define OPTIMIZE
5 #endif
```

un programa con, por ejemplo, el nivel máximo de optimización ofrecido por GCC, el comando de compilación debe ser invocado de la siguiente forma:

```
g++ source.c -o output -Dlevel = '"O3"'
```

De este modo, el compilador solo optimiza las líneas de código contenidas en las funciones a optimizar (con el atributo `OPTIMIZE` en la declaración de la función) y respeta el resto del código.

Para medir los tiempos de ejecución se utiliza el temporizador de la clase *Stopwatch* [174] (ver Código 5.3), con funciones que permiten utilizar el contador de impulsos del reloj (*tick counter*) de la librería *Chrono*, disponible a partir de la versión 11 de C++. Esta opción ha sido la escogida para realizar las mediciones debido a su resolución de alrededor de 10 milisegundos y la ausencia de *overhead* asociado a su ejecución, puesto que la única latencia producida se genera en la llamada al sistema para obtener el tiempo actual y ésta no resulta significativa cuando se cronometran tareas de decenas de milisegundos o más [174]. Además, cabe destacar que este sistema de medición es compatible con otros sistemas operativos.

Antes de ejecutar cada test se realiza una llamada a la función *startTime* de *Stopwatch* para que comience a contar (ver línea 12 del código). Tras realizar las cien millones de ejecuciones del test, se obtiene el tiempo transcurrido con *getTime*, que devuelve el resultado (diferencia entre los tiempos final e inicial) en milisegundos (ver línea 16). Este proceso es repetido diez veces para cada test con objeto de equilibrar las posibles variaciones entre los tiempos de cada ejecución, como se mencionó anteriormente. Tras obtener la media de los diez resultados (expresada también en milisegundos) ésta es dividida por el número de iteraciones, obteniendo el tiempo de una única ejecución, el cual

Código 5.3: Clase Stopwatch utilizada para las mediciones.

```
#include <chrono>
using namespace std::chrono;

class Watchtime{
   std::chrono::system_clock::time_point m_start;
   system_clock::duration diff;
public:
   void startTime();
   unsigned getTime();
};

void Watchtime::startTime(){
   system_clock::time_point::min();       // pone a cero el temporizador
   Watchtime::m_start = std::chrono::system_clock::now();  // lo inicia
}
unsigned Watchtime::getTime(){
   Watchtime::diff = system_clock::now() - m_start;
   return (unsigned)(duration_cast<milliseconds>(diff).count());
}
```

es convertido a nanosegundos para facilitar su lectura. La desviación estándar también es calculada con objeto de cuantificar la dispersión de las mediciones.

Para evitar probables alteraciones originadas por otras tareas ejecutadas en segundo plano por el sistema operativo o por cambios de contexto producidos durante los experimentos, estos son iniciados con llamadas realizadas mediante acceso remoto a los dispositivos RPi empleando el protocolo de red SSH (*Secure Shell*). Además, Raspbian es configurado previamente para mantener en ejecución únicamente los servicios mínimos necesarios, desactivando otros como *lightdm desktop*, las unidades *bluetooth* e inalámbrica y ciertos demonios (*daemons*) como *avahi* o *triggerhappy*.

5.2.1.1.2. Resultados

Los códigos fuente de todos los tests desarrollados *ad hoc* para analizar cada una de las técnicas propuestas pueden ser consultados en el Apéndice C. En muchos casos, reducir los tiempos de ejecución de los programas supone a su vez incrementos en el tamaño del código que pueden afectar a su legibilidad. Aunque todos los experimentos fueron implementados en C++, pueden ser igualmente aplicados a lenguaje C. Asimismo, cabe destacar las innumerables variaciones posibles que podrían haberse aplicado en los distintos experimentos realizados. El objetivo principal de estos es demostrar que las alternativas planteadas consiguen significativas reducciones de los tiempos de ejecución, mejorando notablemente incluso las optimizaciones automáticas que ofrece el compilador cuando estas técnicas no son aplicadas. De este modo se demuestra la necesidad de considerar estas propuestas para el desarrollo de código eficiente.

Los tiempos de ejecución (expresados en nanosegundos, ns) de los tests desarrollados para cada una de las técnicas descritas en el Capítulo 4 son mostrados en las Tablas 5.5,

5.6, 5.7 y 5.8, organizadas según el dispositivo en que se ejecutan los experimentos (RPi 2 modelo B, RPi 3 modelos B y B+ y RPi modelo 4 B).

En ellas se pueden observar los resultados del código estándar y del eficiente (este último obtenido al aplicar la correspondiente técnica sobre el estándar), aplicando o no la optimización automática que ofrece el compilador, es decir, con nivel de optimización 3 y sin optimización (nivel de optimización 0) respectivamente. También se muestra el porcentaje de mejora entre las ejecuciones de ambos códigos para facilitar el análisis de los resultados. Las desviaciones estándar obtenidas han sido mínimas, con una desviación media del 0,31 % considerando los cuatro niveles de optimización (no solo los mostrados en las tablas), asegurando de este modo que todas las medidas tomadas son similares a los promedios extraídos.

Debe tenerse en cuenta que aunque las diferencias respecto al tiempo de ejecución entre un código estándar y su versión optimizada pueden parecer insignificantes, éstas son referidas a una única ejecución, por lo que alcanzan una importancia bastante destacable cuando se ejecutan reiteradamente, por ejemplo, en bucles con numerosas iteraciones. Esto resulta especialmente relevante en programas que se ejecutan 24 horas al día y 7 días a la semana durante todo el año, algo muy común en IoT y este tipo de dispositivos. Además, cabe destacar que las técnicas propuestas no son excluyentes entre sí, siendo posible aplicar conjuntamente varias de ellas sobre un mismo fragmento de código, aumentando así las posibilidades de mejora.

Como era de esperar, los tiempos de ejecución de los códigos compilados con el máximo nivel de optimización (nivel 3) son significativamente menores (aplicando o no las técnicas). Sin embargo, es importante analizar también los efectos de escribir código eficiente sin las optimizaciones ofrecidas por el compilador puesto que, a diferencia del resto de niveles, en el nivel 0 es posible depurar el código (como se ha mencionado anteriormente). En consecuencia, se demuestra de esta forma que el uso de las técnicas propuestas está también especialmente recomendado en casos que requieren intensos procesos de depuración en los cuales no pueden aplicarse las optimizaciones automáticas del compilador. Sin embargo, las tablas no incluyen los resultados obtenidos en relación al resto de niveles (1 y 2) puesto que no son particularmente representativos para los objetivos de esta investigación, al proporcionar valores que no mejoran los resultados. No obstante, los datos completos de todos los experimentos realizados pueden ser consultados en los materiales suplementarios del artículo publicado [237].

Las Figuras 5.6 y 5.7 son proporcionadas para facilitar la comprensión de la información relacionada en las tablas. Ambas gráficas muestran en sus ejes horizontales los números que identifican cada una de las técnicas aplicadas, mientras que en los ejes verticales se ofrecen los porcentajes de mejora alcanzados en los tiempos de ejecución cuando se utiliza cada técnica (respecto a las versiones de código estándar). Asimismo, también son representados los tres modelos de RPi utilizados en los experimentos: 2B (barras amarillas), 3B (barras rojas), 3B+ (barras azules) y 4B (barras verdes). En las ejecuciones representadas por la Figura 5.6 no se utilizan las optimizaciones automáticas ofrecidas por el compilador (es decir se establece el nivel 0 al compilar los tests), mientras que la Figura 5.7 se corresponde con los tests ejecutados con el nivel más alto de optimización.

Tabla 5.5: Tiempos de ejecución y porcentajes de mejora en RPi 2B.

	Técnicas	Tiempos de ejecución (sin optimización)			Tiempos de ejecución (optimización nivel 3)		
		Estándar (ns)	Eficiente (ns)	Mejora (%)	Estándar (ns)	Eficiente (ns)	Mejora (%)
1	Campos de bits	85,71	42,28	50,67	31,25	28,92	7,46
2	Conjuntos de bits	1.707,83	181,42	89,38	1.716,1	182,63	89,36
3	Retorno de booleanos	40,63	38,35	5,61	20,32	20,30	0,10
4	Llamadas en cascada a funciones	828,28	418,60	49,46	514,83	98,46	80,88
5	Acceso por fila principal en matrices	10.826,36	10.835,85	0,00	1.474,66	396,28	73,13
6	Listas de inicialización	1.158,58	1.128,70	2,58	1.148,76	1.107,63	3,58
7	Eliminación de subexpresiones	74,50	55,29	25,79	36,13	34,97	3,21
8	Mapeo de estructuras	522,61	812,45	0,00	505,80	442,10	12,59
9	Eliminación de código muerto	32,47	24,61	24,21	16,82	16,78	0,24
10	Control de excepciones	9.465,32	42,51	99,55	9.406,65	10,06	99,89
11	Variables globales en bucles	558,62	425,40	23,85	87,27	87,29	0,00
12	Funciones inline o insertadas	55,66	35,60	36,04	20,15	20,13	0,10
13	Variables globales	1.538,84	1.211,68	21,26	697,21	589,75	15,41
14	Constantes en bucles	504,34	353,86	29,84	224,77	228,08	0,00
15	Inicialización frente a asignación	57,09	27,96	51,02	54,55	10,01	81,65
16	División con potencias de dos	35,83	35,79	0,11	20,16	20,13	0,15
17	Mutiplicación con potencias de dos	35,63	35,60	0,08	20,05	20,02	0,15
18	Integer frente a Character	60,45	54,82	9,31	20,14	20,13	0,05
19	Bucles con cuenta regresiva	1.483,21	1.824,74	0,00	360,35	360,29	0,02
20	Desenrollado de bucles	770,95	501,29	34,98	115,28	78,31	32,07
21	Paso de estructuras por referencia	509,51	44,75	91,22	474,79	10,06	97,88
22	Aliasing de punteros	121,96	114,11	6,44	71,63	68,24	4,73
23	Cadenas de punteros	84,58	61,19	27,65	24,64	24,61	0,12
24	Pre-incremento vs. post-incremento	2.706,01	2.707,06	0,00	692,16	690,97	0,17
25	Búsqueda lineal	2.487,52	2.060,83	17,15	700,99	378,30	46,03
26	Estructuras IF en bucles	2.161,82	1.496,00	30,80	156,92	156,88	0,03

Tabla 5.6: Tiempos de ejecución y porcentajes de mejora en RPi 3B.

	Técnicas	Tiempos de ejecución (sin optimización)			Tiempos de ejecución (optimización nivel 3)		
		Estándar (ns)	Eficiente (ns)	Mejora (%)	Estándar (ns)	Eficiente (ns)	Mejora (%)
1	Campos de bits	55,09	28,35	48,54	21,05	20,16	4,23
2	Conjuntos de bits	1.111,43	124,3	88,82	1.112,33	124,83	88,78
3	Retorno de booleanos	32,84	25,23	23,17	12,65	12,61	0,32
4	Llamadas en cascada a funciones	532,65	295,25	44,57	309,11	70,57	77,17
5	Acceso por fila principal en matrices	7.067,18	7.062,65	0,06	933,55	136,77	85,35
6	Listas de inicialización	766,16	746,05	2,62	757,83	731,68	3,45
7	Eliminación de subexpresiones	50,05	37,52	25,03	26,20	25,22	3,74
8	Mapeo de estructuras	399,75	524,69	0,00	381,80	371,36	2,73
9	Eliminación de código muerto	21,03	15,12	28,10	10,96	10,92	0,36
10	Control de excepciones	5.465,30	26,68	99,51	5.460,63	6,67	99,88
11	Variables globales en bucles	382,89	281,08	26,59	74,11	74,07	0,05
12	Funciones inline o insertadas	37,88	23,54	37,86	12,54	12,50	0,32
13	Variables globales	910,18	749,08	17,70	521,35	480,29	7,88
14	Constantes en bucles	340,73	236,00	30,74	144,84	137,48	5,08
15	Inicialización frente a asignación	40,35	18,48	54,20	37,03	6,72	81,85
16	División con potencias de dos	23,41	23,35	0,26	12,55	12,51	0,32
17	Multiplicación con potencias de dos	23,47	23,35	0,51	12,64	12,61	0,24
18	Integer frente a Character	40,07	35,86	10,51	12,64	12,60	0,32
19	Bucles con cuenta regresiva	1.027,85	1.292,02	0,00	269,68	188,72	30,02
20	Desenrollado de bucles	532,41	330,48	37,93	93,45	57,18	38,81
21	Paso de estructuras por referencia	348,64	30,02	91,39	332,00	6,72	97,98
22	Aliasing de punteros	84,31	78,39	7,02	48,47	45,86	5,38
23	Cadenas de punteros	60,54	43,69	27,83	16,82	16,79	0,18
24	Pre-incremento vs. post-incremento	1.867,60	1.868,37	0,00	519,81	519,78	0,01
25	Búsqueda lineal	1.690,00	1.368,81	19,01	435,44	284,90	34,57
26	Estructuras IF en bucles	1.538,42	1.038,22	32,51	123,54	123,50	0,03

Tabla 5.7: Tiempos de ejecución y porcentajes de mejora en RPi 3B+.

	Técnicas	Tiempos de ejecución (sin optimización)			Tiempos de ejecución (optimización nivel 3)		
		Estándar (ns)	Eficiente (ns)	Mejora (%)	Estándar (ns)	Eficiente (ns)	Mejora (%)
1	Campos de bits	47,92	24,67	48,52	18,18	17,41	4,24
2	Conjuntos de bits	951,22	106,57	88,80	953,78	106,56	88,83
3	Retorno de booleanos	28,35	21,77	23,21	10,90	10,88	0,18
4	Llamadas en cascada a funciones	460,35	256,04	44,38	267,48	61,58	76,98
5	Acceso por fila principal en matrices	6.057,46	6.052,24	0,09	794,92	116,79	85,31
6	Listas de inicialización	673,66	667,99	0,84	662,28	653,39	1,34
7	Eliminación de subexpresiones	43,01	32,16	25,23	22,46	21,52	4,19
8	Mapeo de estructuras	338,54	460,12	0,00	324,79	323,35	0,44
9	Eliminación de código muerto	17,94	12,86	28,32	9,50	9,43	0,74
10	Control de excepciones	4.988,09	23,59	99,53	5.056,53	6,03	99,88
11	Variables globales en bucles	330,50	243,72	26,26	63,54	63,81	0,00
12	Funciones inline o insertadas	32,21	20,01	37,88	10,93	10,88	0,46
13	Variables globales	793,95	679,45	14,42	464,86	436,42	6,12
14	Constantes en bucles	289,83	202,27	30,21	126,07	119,64	5,10
15	Inicialización frente a asignación	34,35	15,72	54,24	31,52	5,71	81,88
16	División con potencias de dos	20,08	20,01	0,35	10,93	10,88	0,46
17	Multiplicación con potencias de dos	20,07	20,01	0,30	10,80	10,72	0,74
18	Integer frente a Character	34,88	31,20	10,55	10,77	10,72	0,46
19	Bucles con cuenta regresiva	889,60	1.103,27	0,00	240,11	180,32	24,90
20	Desenrollado de bucles	456,63	286,43	37,27	79,51	48,60	38,88
21	Paso de estructuras por referencia	301,95	26,29	91,29	285,03	5,85	97,95
22	Aliasing de punteros	72,21	67,18	6,97	41,57	39,31	5,44
23	Cadenas de punteros	51,51	37,16	27,86	14,34	14,29	0,35
24	Pre-incremento vs. post-incremento	1.611,73	1.652,95	0,00	469,88	486,87	0,00
25	Búsqueda lineal	1.455,65	1.177,63	19,10	386,03	270,78	29,86
26	Estructuras IF en bucles	1.321,56	895,18	32,26	105,90	105,86	0,04

Tabla 5.8: Tiempos de ejecución y porcentajes de mejora en RPi 4B.

	Técnicas	Tiempos de ejecución (sin optimización)			Tiempos de ejecución (optimización nivel 3)		
		Estándar (ns)	Eficiente (ns)	Mejora (%)	Estándar (ns)	Eficiente (ns)	Mejora (%)
1	Campos de bits	24,99	10,36	58,54	7,39	7,48	0,00
2	Conjuntos de bits	953,95	81,44	91,46	953,99	81,44	91,46
3	Retorno de booleanos	11,37	8,73	23,22	4,73	4,73	0,00
4	Llamadas en cascada a funciones	193,76	218,22	0,00	189,53	19,41	89,76
5	Acceso por fila principal en matrices	5.338,36	4.637,71	13,12	651,93	140,84	78,40
6	Listas de inicialización	238,43	232,74	2,39	373,38	331,64	11,18
7	Eliminación de subexpresiones	16,22	15,28	5,80	7,06	8,08	0,00
8	Mapeo de estructuras	234,92	296,93	0,00	218,25	228,25	0,00
9	Eliminación de código muerto	6,17	4,76	22,85	4,78	4,53	5,23
10	Control de excepciones	2.474,97	11,4	99,54	2.420,49	5,32	99,78
11	Variables globales en bucles	126,23	113,24	10,29	34,78	34,75	0,09
12	Funciones inline o insertadas	14,47	8,07	44,23	4,73	4,75	0,00
13	Variables globales	326,97	294,94	9,80	232,27	214,88	7,49
14	Constantes en bucles	102,31	86,13	15,81	60,12	59,44	1,13
15	Inicialización frente a asignación	13,73	6,41	53,31	13,72	4,76	65,31
16	División con potencias de dos	7,37	8,06	0,00	4,71	4,96	0,00
17	Multiplicación con potencias de dos	7,39	8,04	0,00	4,73	5,27	0,00
18	Integer frente a Character	16,84	15,42	8,43	4,73	4,73	0,00
19	Bucles con cuenta regresiva	1.023,09	1.024,83	0,00	148,85	147,52	0,89
20	Desenrollado de bucles	524,45	249,59	52,41	37,42	34,73	7,19
21	Paso de estructuras por referencia	102,83	11,53	88,79	95,57	5	94,77
22	Aliasing de punteros	26,78	27,54	0,00	18,73	16,1	14,04
23	Cadenas de punteros	20,8	24,07	0,00	6,72	6,93	0,00
24	Pre-incremento vs. post-incremento	2.022,26	2.022,42	0,00	283,6	281,64	0,69
25	Búsqueda lineal	622,02	434,35	30,17	167,54	155,48	7,20
26	Estructuras IF en bucles	1.089,09	1.024,15	5,96	44,81	44,77	0,09

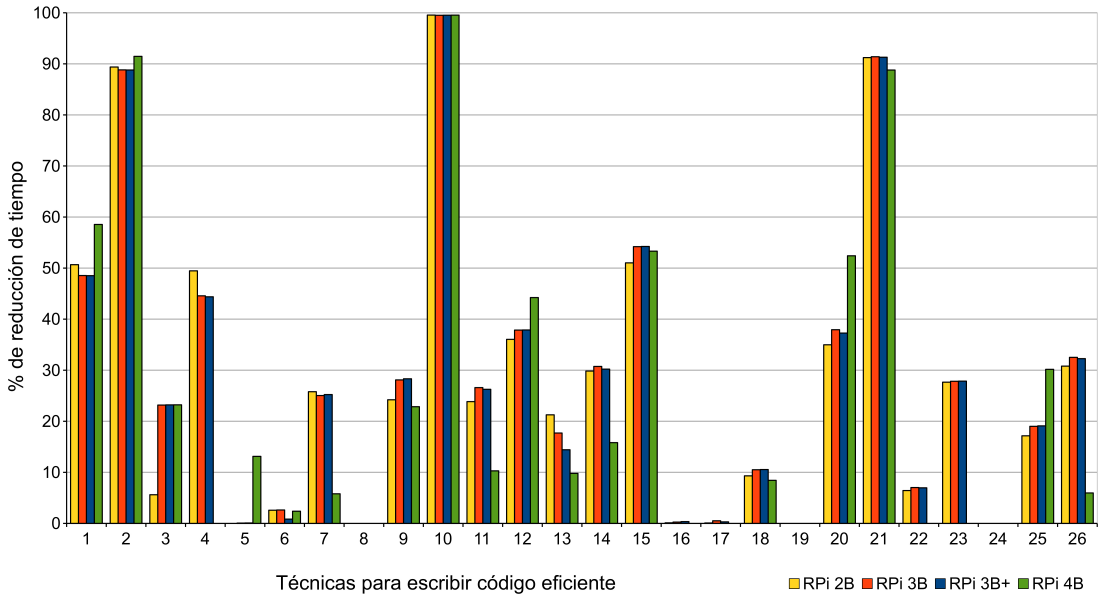

Figura 5.6: Porcentajes de reducción de tiempos de ejecución obtenidos mediante la escritura de código eficiente (sin optimización del compilador) sobre dispositivos IoT.

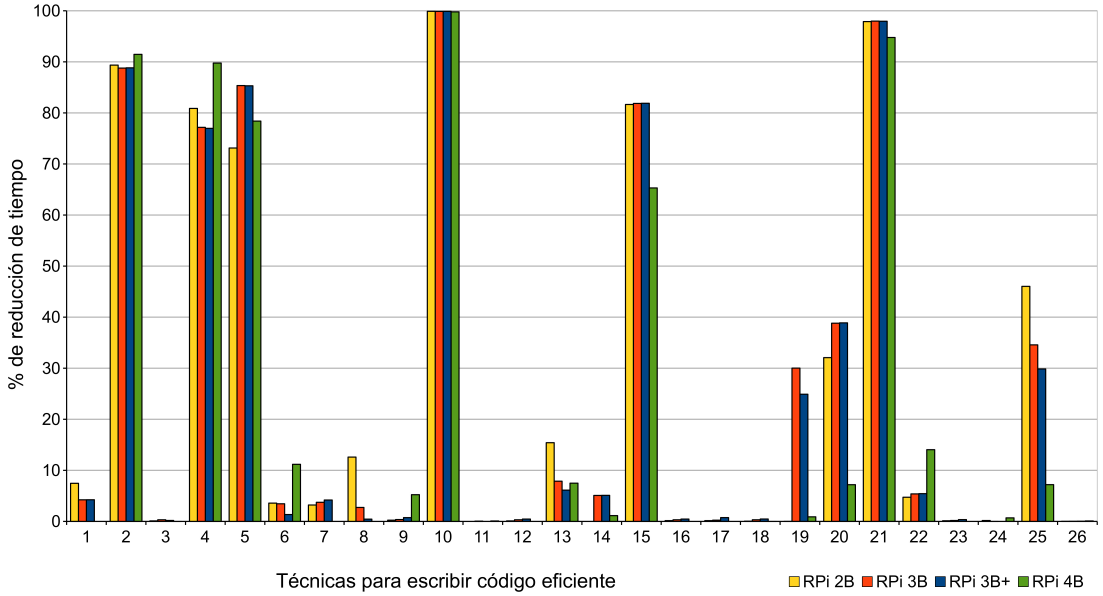

Figura 5.7: Porcentajes de reducción de tiempos de ejecución obtenidos mediante la escritura de código eficiente (con nivel 3 de optimización del compilador) sobre dispositivos IoT.

Las técnicas permiten conseguir ahorros de tiempo de hasta el 99,89 %, como se observa en las figuras y tablas. Especialmente destacables son los resultados (analizados más adelante) obtenidos con las técnicas de *conjuntos de bits* (T2), *control de excepciones* (T10) y *paso de estructuras por referencia* (T21), lo cual era esperable en las dos últimas al ser ambas ampliamente conocidas.

Como muestran las tablas, cuanto más avanzado es el modelo de Raspberry Pi, más reducidos son los tiempos de ejecución. En general, los porcentajes de mejora son similares en los cuatro modelos de RPi, aunque existen algunas diferencias significativas. Así, en relación al nivel de optimización 0 (ver Figura 5.6), las siguientes técnicas logran un mayor ahorro de tiempo con el modelo RPi 4 (hasta 17 puntos porcentuales más): *campos de bits* (T1); *acceso por fila principal en matrices* (T5); *funciones inline o insertadas* (T12); *desenrollado de bucles* (T20); y *búsqueda lineal* (T25). Sin embargo, con el nivel de optimización 3 (ver Figura 5.7) es el modelo RPi 2B el que destaca especialmente en ciertos casos, con diferencias hasta un 39 % superiores en las siguientes técnicas: *mapeo de estructuras* (T8); *uso de variables globales* (T13) y búsqueda lineal (T25). Esto no significa que sus tiempos de ejecución sean menores, sino que el impacto de aplicar las técnicas es superior en este modelo.

En general, con el nivel 0 (Figura 5.6) se consiguen mejoras en la mayoría de los casos excepto en siete, en los que los códigos eficientes obtienen prácticamente los mismos tiempos de ejecución que los códigos estándar. En concreto, no se demuestra la eficacia de las siguientes técnicas, a pesar de haber realizado sustanciales esfuerzos para intentar demostrar su utilidad: *acceso por fila principal en matrices* (T5), excepto con RPi 4; *listas de inicialización* (T6); *mapeo de estructuras* (T8); *división con denominadores potencias de 2* (T16); *multiplicación con factores potencias de 2* (T17); *bucles con cuenta regresiva* (T19) y *pre-incremento frente a pos-incremento* (T24). De hecho, descartando estas técnicas el resto consiguen una mejora media del 38,97 %. De este modo, se demuestra la eficiencia de 19 técnicas para escribir código eficiente con la optimización por defecto del compilador (es decir, con nivel 0) en dispositivos IoT.

Mediante la utilización del nivel 3 (Figura 5.7) el compilador resuelve con éxito la mayoría de los casos. Aun así, como se puede observar, es recomendable escribir código eficiente de forma manual en relación a las siguientes nueve técnicas, con cuya aplicación se alcanza un porcentaje de mejora medio en los tiempos de ejecución del 68,88 %: *conjuntos de bits* (T2); *llamadas en cascada a funciones* (T4); *acceso por fila principal en matrices* (T5); *control de excepciones* (T10); *inicialización frente a asignación* (T15); *bucles con cuenta regresiva* (T19); *desenrollado de bucles* (T20); *paso de estructuras por referencia* (T21) y *búsqueda lineal* (T25).

A continuación, estas nueve técnicas son analizadas con mayor profundidad, evaluando los factores que condicionan su mejora, pero considerando únicamente los modelos RPi 3B+ y 4 por ser los dispositivos que logran mejores tiempos de ejecución (como se esperaba, dadas sus características). Además, el modelo 3B+ permanecerá en producción al menos hasta enero de 2026, y el RPi 4 ha sido el último en ser lanzado (septiembre de 2019).

Conjuntos de bits (T2)

Los experimentos realizados demuestran que es recomendable utilizar conjuntos de *bits* en lugar de vectores cuando se requiere trabajar con más de cinco elementos booleanos, debido a que estos conjuntos optimizan el espacio y los accesos a los *bits* son más rápidos. El resumen de las implementaciones desarrolladas para ambos tests puede verse en el Código 5.4.

La versión estándar consiste en un vector booleano recorrido mediante un bucle que se encarga de establecer cada uno de sus elementos a `true`, mientras que en el código eficiente se define un conjunto de *bits* y una única llamada a la función `set` es suficiente para poner a uno todos sus *bits*. Además, cabe destacar que los elementos individuales son accedidos como tipos de referencias especiales, dado que en la mayoría de entornos C/C++ no existe ningún tipo elemental que represente un único *bit*.

En la Figura 5.8 puede observarse la variación del porcentaje de mejora en función del número de elementos booleanos requeridos. Las pruebas realizadas demuestran que no se obtiene ninguna ventaja cuando esta técnica es aplicada sobre un número inferior a 6 elementos. Sin embargo, con solo 10 el porcentaje de reducción del tiempo de ejecución se sitúa alrededor del 50 %. En el caso de RPi 4, con 30 booleanos la mejora es del 85,11 % (77,17 % en la RPi 3B+), mientras que con 100 elementos la ventaja supera el 91 % (89 % en RPi 3B+), estabilizándose alrededor del 95 % en ambos dispositivos a partir de 250 elementos.

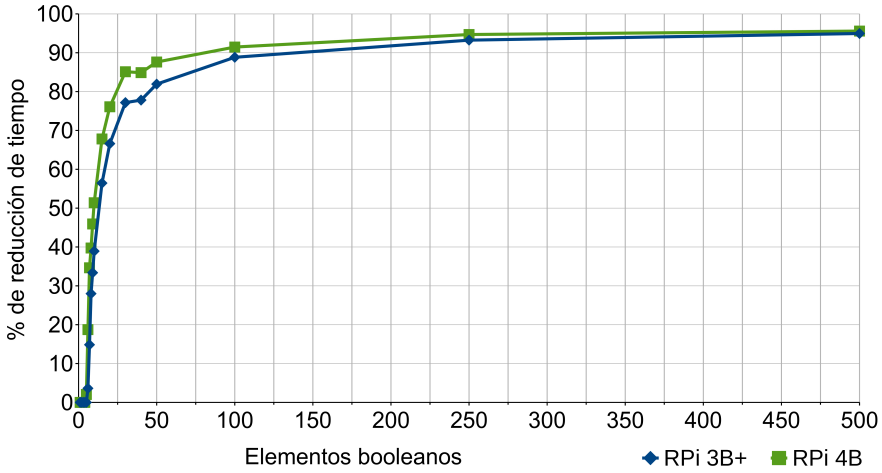

Figura 5.8: Conjuntos de bits (T2). Porcentajes de mejora en los tiempos de ejecución según el número de elementos booleanos requeridos en RPi 3B+ y 4B.

Código 5.4: Conjuntos de bits (T2). Resumen comparativo de los tests.

```
1 void standardCode() {
2     bool array[100];
3     for (int i=0; i<32; i++)
4         array[i] = true;
5 }
```

```
void efficientCode() {
    std::bitset<100> bitset;
    bitset.set();
}
```

Llamadas en cascada a funciones (T4)

En esta técnica el código estándar implementa un bucle que comprueba en cada iteración el contenido de una variable de tipo *Integer* (ver en el Código 5.5 un resumen comparativo de la implementación de ambos tests). El valor de esta variable es devuelto por la función "`getStatus`", llamada a través de un puntero. En el código eficiente, el retorno de la función es almacenado en una variable auxiliar "`status`", de modo que la función solo es llamada en una ocasión (fuera del bucle) y es la variable auxiliar la que se comprueba en cada una de las iteraciones. Esta técnica puede ser aplicada en aquellos casos en los que el programador (a diferencia del compilador) sabe que el valor de la variable no cambia durante la ejecución del bucle.

La Figura 5.9 muestra cómo varía el porcentaje de mejora en función del número de llamadas a la función, es decir, el número de iteraciones del bucle, con resultados similares en ambos dispositivos, aunque ligeramente superiores en RPi 4. Una única llamada consigue una pequeña disminución de tiempo del 0,49 %, mientras que con dos llamadas ya alcanza el 35 %. Utilizar la alternativa eficiente da como resultado una mejora de más del 70 % en solo 15 iteraciones del bucle ($N = 15$). De 100 a más llamadas, el porcentaje crece muy lentamente, estabilizándose en torno a una reducción del 90 %. En consecuencia, se demuestra que es aconsejable aplicar esta técnica siempre que sea posible bajo las condiciones mencionadas.

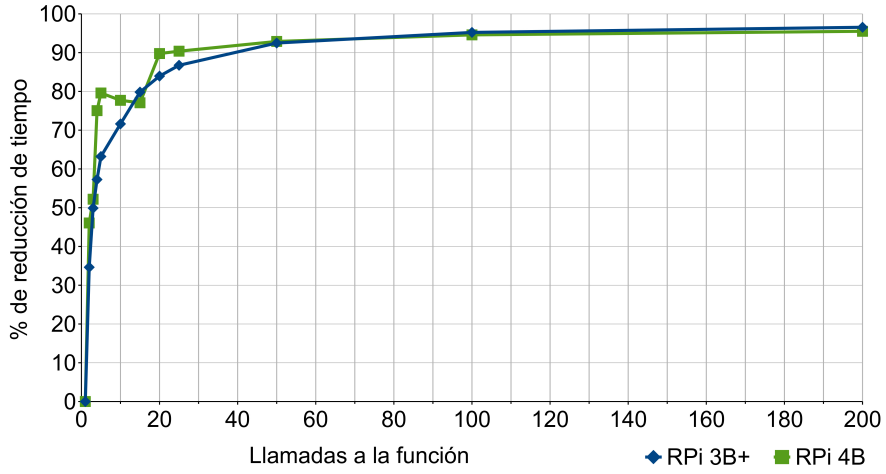

Figura 5.9: Llamadas en cascada a funciones (T4). Porcentajes de mejora en los tiempos de ejecución en función del número de llamadas a la función (número de iteraciones del bucle) en RPi 3B+ y 4B.

Código 5.5: Llamadas en cascada a funciones (T4). Resumen comparativo de los tests.

```
1  void standard(Test *test){
2    for (int i=0; i<N; i++)
3      if (test->getStatus()==1)
4        a[i] = 0;
5  }
6
```

```
void efficient(Test *test){
  int status = test->getStatus();
  for (int i=0; i<N; i++)
    if (status==1)
      a[i] = 0;
}
```

Acceso por fila principal en matrices (T5)

Esta técnica se fundamenta en acceder al contenido de las matrices en el orden en que sus elementos son almacenados, con objeto de lograr mejores tasas de aciertos en caché. El código estándar recorre el vector iterando sobre cada columna antes de pasar a la siguiente (procesando las columnas en el bucle externo y las filas en el interno), mientras que el código eficiente atraviesa la matriz priorizando por filas (ver Código 5.6).

La Figura 5.10 muestra cómo crece el porcentaje de mejora a medida que aumenta el tamaño de la matriz (a mayor tamaño, mayor porcentaje de mejora), alcanzando hasta un 85,31 % en el modelo 3B+ y 78,40 % en el 4. Estas mejoras están originadas principalmente por el incremento de trabajo de la caché conforme crecen las dimensiones de la matriz y por la reducción de eficiencia producida cuando se usa la iteración por columna principal.

También es destacable la diferencia entre los resultados de ambos modelos, debido a que la arquitectura del Cortex-A72 del modelo RPi 4 cuenta con una caché mayor que conduce a una reducción de la penalización por error al aplicar el código estándar. Así, a partir de 40 o más elementos el porcentaje se estabiliza en torno a una mejora del 81 % en RPi 3B+ y del 78,40 % en RPi 4. Este hecho también influye notablemente en los resultados obtenidos con un tamaño de matriz de 30 elementos, donde se incrementa la tasa de aciertos de caché.

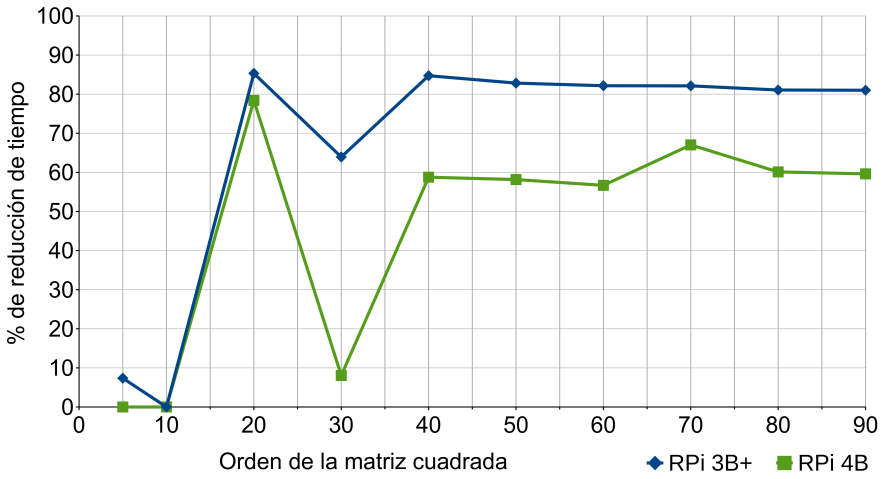

Figura 5.10: Acceso por fila principal en matrices (T5). Porcentajes de mejora en los tiempos de ejecución según el orden de la matriz cuadrada en RPi 3B+ y 4B.

Código 5.6: Acceso por fila principal en matrices (T5). Resumen comparativo de los tests.

```
1 int standard(){
2   for (int j=0; j<N; j++)
3     for (int i=0; i<N; i++)
4       array[i][j] = 0;
5 }
```

```
int efficient(){
  for (int i=0; i<N; i++)
    for (int j=0; j<N; j++)
      array[i][j] = 0;
}
```

Este experimento emplea matrices cuadradas, con el mismo número de filas y columnas. Sin embargo, la mejora también estaría influida por la relación de tamaño entre sus dimensiones. Por ejemplo, al recorrer una matriz estrecha (con pocas columnas) por orden de columna principal, se pueden encontrar filas consecutivas en la caché y, por tanto, los resultados obtenidos al ejecutar el código estándar mejorarían sustancialmente. Por este motivo, se prevé abordar próximamente el análisis de diversas variaciones centradas en los tamaños de ambas dimensiones.

Control de excepciones (T10)

Esta técnica, basada en modificar el uso de excepciones en bucles (ver Código 5.7), obtiene un grado de mejora que depende directamente del número de ellas que son procesadas. En la prueba estándar del experimento se lanzan 100 excepciones simples, sin ningún tipo de contenido salvo su declaración:

$$class\ myexception:\ public\ exception\ \{\}\ myex;$$

Considerando esta implementación, la utilización de la versión eficiente alcanza una mejora del 99 %, debido a las penalizaciones de tiempo en las que se incurre al utilizar el controlador de excepciones (detallas en el apartado T10 del Capítulo 4). De este modo se demuestra que, aunque las excepciones son necesarias para soportar el control de errores, es recomendable utilizar esta técnica cuando dichas excepciones están incluidas dentro de bucles y existe una probabilidad relativamente alta de que ocurran, siendo posible gestionarlas y evitar su lanzamiento.

Por tanto, dado que el grado de mejora depende directamente del número de excepciones lanzadas y de la estructura de datos empleada, podrían aplicarse numerosas variaciones para analizar con mayor profundidad esta técnica. Por ello, en este caso, se omite la evaluación detallada de las condiciones que influyen en este rendimiento, que será abordada más adelante, puesto que además no existen diferencias significativas entre los resultados obtenidos por los cuatro modelos de RPi (como se observaba en la Figura 5.7 -pág. 87-).

Código 5.7: Control de excepciones (T10). Resumen comparativo de los tests.

```
int standard(){
  int num = 100;
  for (int i=0; i<1; i++){
    try{
      if (num == 100) {
        throw myex;
      }
    } catch (exception& e){
    }
  }
  return 0;
}
```

```
int efficient(){
  int num = 100;
  for (int i=0; i<1; i++){
    if (num != 100) {
      continue;
    }
  }
  return 0;
}
```

Inicialización frente a asignación (T15)

Para analizar esta técnica se utilizan números complejos con partes real e imaginaria de doble precisión. En el código eficiente, la variable correspondiente es inicializada directamente al ser declarada, mientras que en el código estándar su valor es establecido mediante una asignación posterior independiente a la declaración (ver Código 5.8). En este caso, el porcentaje de mejora conseguido con el código eficiente alcanza un 81,88 % en RPi 3B y un 65,31 % en RPi 4, al evitar la asignación adicional de la instanciación del código estándar. Aunque en general se recomienda utilizar esta técnica indistintamente del tipo de datos empleado, se espera analizar de forma pormenorizada en próximos trabajos su aplicación con otros tipos y estructuras.

Bucles con cuenta regresiva (T19)

Esta técnica es aplicable en aquellos bucles en los que el orden seguido por su iterador no es determinante (ver T19 en el Capítulo 4). En el código estándar el bucle se recorre hacia adelante mediante un bucle `for`, mientras que en la versión eficiente éste es transitado en sentido contrario con una estructura de tipo `while` (ver Código 5.9).

En el modelo RPi 3B este cambio se traduce en mejoras en los tiempos de ejecución de entre el 20 % y el 30 % con vectores de tamaño superior a 20 elementos, debido al mayor número de operaciones requeridas para atravesar el bucle hacia delante. Sin embargo, en el modelo 4 la mejora es prácticamente nula, con un ahorro de tiempo máximo del 1,93 % (ver Figura 5.11).

Para entender este comportamiento es importante destacar que en el modelo RPi 4 los tiempos de ejecución son significativamente menores, de forma que incluso el código estándar en este dispositivo es hasta un 50 % más rápido que el eficiente en el dispositivo 3B+. En consecuencia, las posibilidades de mejorar aún más los resultados mediante la utilización de código eficiente se reducen significativamente. Esto es debido no solo a la

Código 5.8: Inicialización frente a asignación (T15). Resumen comparativo de los tests.

```
1 void standard() {                          void efficient() {
2   std::complex<double> mycomplex;            std::complex<double> mycomplex(3.14);
3   mycomplex = (3.14);                      }
4 }
```

Código 5.9: Bucles con cuenta regresiva (T19). Resumen comparativo de los tests.

```
1 void standard(){                           void efficient() {
2   for (int i=0; i<N; i++) {                  int i = N+1;
3     a[i]=i;                                  while (--i) {
4   }                                            a[i] = i;
5 }                                            }
6                                            }
```

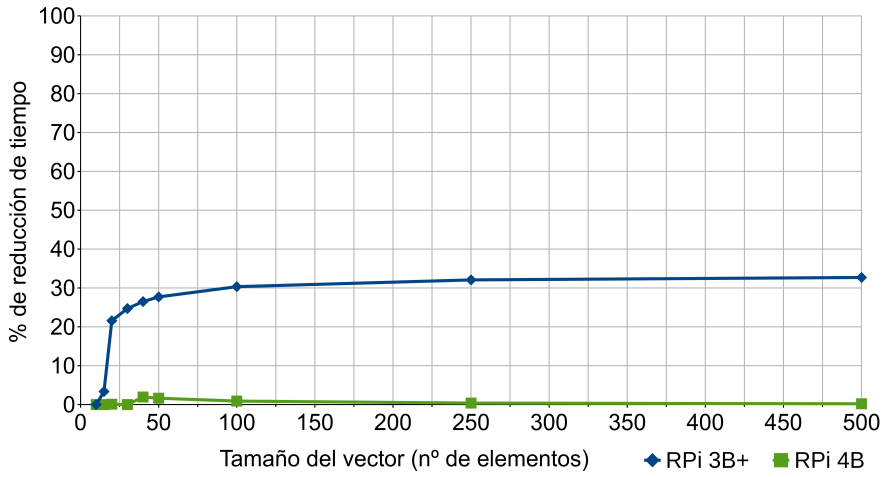

Figura 5.11: Bucles con cuenta regresiva (T19). Porcentajes de mejora en el tiempo de ejecución según el tamaño del vector en RPi 3B+ y 4B.

ejecución fuera de orden[8] (OoOE, *Out-of-Order Execution*) y la mayor caché del modelo 4 (que conduce a una reducción en la penalización por fallo (*miss penalty*) al aplicar el código estándar, disminuyendo así la diferencia con el código de eficiencia energética), sino también por las mejoras específicas introducidas en el Cortex-A72 de este dispositivo[9] que suponen una mejora del rendimiento de entre el 10 % y el 50 % según ARM [241], especialmente en lo que respecta a las cargas de trabajo de punto flotante.

Desenrollado de bucles (T20)

Los tests ejecutados para analizar la eficiencia de la técnica de desenrollado de bucles se centran en inicializar los elementos de un vector de enteros. En el código estándar se realiza una única asignación (inicialización a cero) en cada iteración del bucle (ver Código 5.10), mientras que en la versión eficiente se desenrolla éste y se realizan varias operaciones de asignación en cada iteración, aumentando así el tamaño del código, pero reduciendo el número de iteraciones.

El rendimiento de esta técnica depende tanto del tamaño del vector como del número de operaciones desenrolladas. Con el objetivo de facilitar la comprensión de los resultados de los experimentos adicionales realizados se ofrece la Figura 5.12, relativa a la aplicación de la técnica con vectores de 50, 100, 200 y 300 elementos. En ella se representan los porcentajes de mejora alcanzados en función del número de iteraciones. Así, tomando como ejemplo el vector de 100 elementos, se necesitan 50 operaciones de asignación para su inicialización completa con dos únicas iteraciones del bucle, 25 asignaciones son necesarias con 4 iteraciones y así sucesivamente, hasta alcanzar las 50 iteraciones, que requieren únicamente dos operaciones de asignación.

[8] El Cortex A72 de los modelos RPi 4B posee la capacidad de ejecutar instrucciones fuera de orden [241], a diferencia del resto de modelos, con Cortex A53, que carecen de esta característica. Esto posibilita que el procesador pueda adelantar la ejecución de ciertas instrucciones, aprovechando los huecos de tiempo dejados por otras anteriores que aún no están listas para su ejecución debido, por ejemplo, a que la información necesaria para realizar una operación aún no se encuentra disponible [242].

[9] En el Cortex-A72 se añaden optimizaciones de potencia y rendimiento adicionales a las existentes en el diseño A53 del resto de modelos RPi, incluyendo un algoritmo de predicción mejorado, mayor ancho de banda, unidades de ejecución de menor latencia y mayor ancho de banda de caché L2.

Código 5.10: Desenrollado de bucles (5.10). Resumen comparativo de los tests.

```
1  void standard(){
2    int i;
3    for (i=0; i<N; i++){
4      array[i] = 0;
5    }
6  }
7
8
9
10
```

```
void efficient(){
  int i;
  for (i=0; i<N; i+=5){
    array[i]   = 0;
    array[i+1] = 0;
    array[i+2] = 0;
    array[i+3] = 0;
    array[i+4] = 0;
  }
}
```

Los resultados en la RPi 4 revelan la ineficacia de la técnica en este modelo, donde únicamente se obtienen disminuciones de tiempo entre un 14,18 % y un 17.57 % al emplear vectores de 50 elementos con un máximo de 5 iteraciones. Respecto al modelo RPi 3B, como se puede observar en la gráfica, cuanto más pequeño es el tamaño del vector, mayor es la mejora alcanzada con esta técnica. Así, no se producen reducciones notables en los tiempos de ejecución cuando se emplea el desenrollado para inicializar matrices de 200 o más elementos (ver Figuras 5.12 c y d). Respecto al vector de 50 elementos (Figura 5.12 a), se obtiene una mejora de aproximadamente el 40 % cuando el número de iteraciones es igual o inferior a 10 (es decir, con más de 5 operaciones de asignación desenrolladas). Al superar las 20 iteraciones (Figuras 5.12 b, c y d) la mejora tiende a desaparecer en los cuatro vectores, de 50, 100, 200 y 300 elementos.

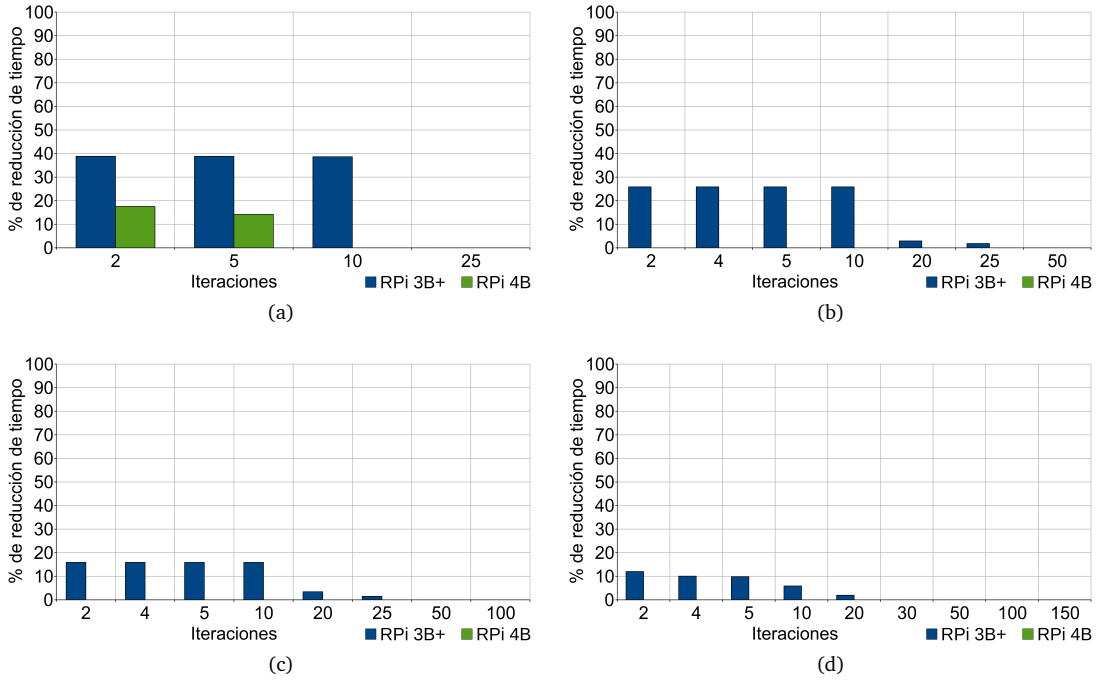

Figura 5.12: Desenrollado de bucles (5.10) en RPi 3B+. Porcentajes de mejora en los tiempos de ejecución según el número de iteraciones. (**a**) Vector de 50 elementos. (**b**) Vector 100 elementos. (**c**) Vector de 200 elementos. (**d**) Vector de 300 elementos.

Paso de estructuras por referencia (T21)

En el código estándar de esta técnica se pasa una estructura por valor y se accede directamente al contenido de una variable entera (ver Código 5.11), mientras que en la versión eficiente la estructura es pasada por referencia y la variable es accedida a través de un puntero ($reference\text{->}getIndex()$). En ambos casos, la estructura consta de una clase con dos miembros de tipo *string* y una subestructura formada por un vector de 10 elementos y una variable entera que sirve como índice. La clase del constructor consiste en tres asignaciones (correspondientes a la inicialización de los dos atributos de tipo *string* y el índice de la subestructura).

Así, pasar la estructura por referencia supone reducciones del 97,95 % (RPi 3B+) y 94,77 % (RPi 4) en el tiempo de ejecución, debido al *overhead* asociado a la copia completa de cada estructura, constructor y el destructor incluidos, que se realiza cuando un parámetro es pasado por valor. Consecuentemente, al igual que sucede con otras técnicas, el porcentaje de ahorro depende de las estructuras de datos utilizadas, existiendo infinidad de posibilidades y debiendo profundizar en su análisis en posteriores estudios.

Búsqueda lineal (T25)

Aunque existen algoritmos similares que requieren menos tiempo de ejecución, el uso de la búsqueda lineal está especialmente recomendado en listas de pocos elementos o para usuarios noveles en programación. Los tests desarrollados consisten en encontrar

Código 5.11: Paso de estructuras por referencia (T21). Resumen comparativo de los tests.

```
1  typedef struct {int array[10]; int index;} Structure;
2
3  class Class {
4    private:
5      string    attribute_a ;
6      string    attribute_b;
7      Structure structure;
8    public:
9      Class(string attribute1, string attribute2, int i);
10     int getIndex();
11 };
12
13 Class::Class(string attribute1, string attribute2, int i) {
14   attribute_a = attribute1;
15   attribute_b = attribute2;
16   structure.index = i;
17 }
18
19 int Class::getIndex() {
20   return structure.index;
21 }
22
23 int standard(Class value){                int efficient(Class *reference){
24   return value.getIndex();                   return reference->getIndex();
25 }                                         }
```

un número dentro de un vector de enteros (ver Código 5.12). En la versión estándar se realiza una búsqueda lineal, de modo que se verifica secuencialmente cada elemento hasta encontrar una coincidencia (devolviendo el índice correspondiente) o hasta que se han recorrido todos los elementos sin encontrar el número buscado (es decir, el valor indicado no coincide con ningún elemento y la función devuelve "-1"). En consecuencia, se necesitan dos comparaciones: una para controlar las iteraciones del bucle y otra para determinar si el elemento de la posición actual coincide con el valor. Por contra, en el código eficiente se emplea una instrucción `while` que realiza una única comparación en cada iteración, tal y como se detalla en la descripción de la técnica (T25) en el Capítulo 4. En ambos tests los vectores son previamente inicializados de forma que el elemento buscado se encuentra siempre en la última posición.

La utilización de esta técnica alcanza en la placa 3B+ mejoras superiores al 22 %, con vectores de seis o más elementos, y en torno al 35 % en vectores cuyo tamaño supera los 100 elementos, como se muestra en la Figura 5.13. En el modelo 4B, sin embargo, los valores máximos se sitúan alrededor del 7 %. En este caso el tiempo de ejecución también es significativamente menor en el modelo RPi 4, de forma que su código estándar es hasta un 50 % más rápido que el eficiente en la RPi 3B+. Así, al igual que se discute en el caso de la técnica de *bucles con cuenta regresiva*, este hecho reduce notablemente las posibilidades de mejorar aún más los resultados mediante la implementación de código eficiente.

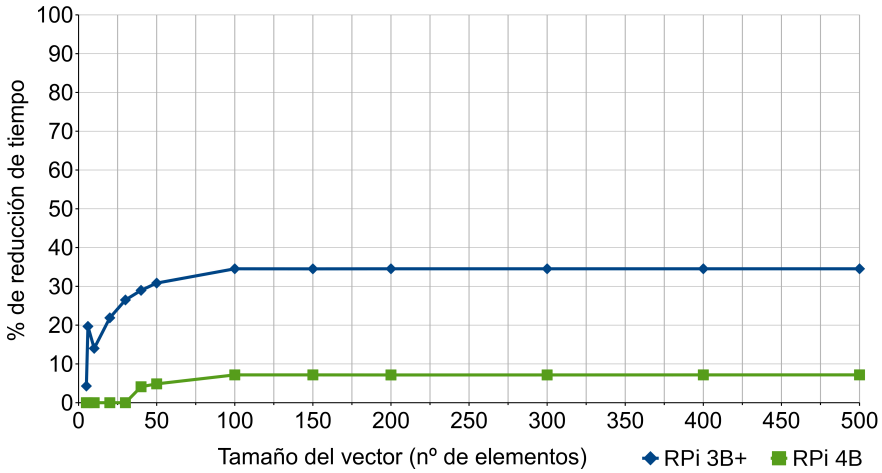

Figura 5.13: Búsqueda lineal (T25). Porcentajes de mejora en el tiempo de ejecución según el tamaño del vector (número de elementos) en RPi 3B+.

Código 5.12: Búsqueda lineal (T25). Resumen comparativo de los tests.

```
1  int std(int *list,int N,int search){
2    int i;
3    for (i = 0; i < N; i++)
4      if (list[i] == search)
5        return i;
6      return -1;
7  }
8
9
```

```
int eff(int *list,int N,int search){
  int i;
  list[N] = search;
  i = 0;
  while (list[i] != search)
    i++;
  if (i == N)
    return -1;
}
```

En resumen, mediante la aplicación de las técnicas propuestas es posible conseguir importantes reducciones del tiempo de ejecución en los cuatro modelos de Raspberry Pi empleados. Además, en algunos de los experimentos realizados (como los correspondientes a las técnicas de *acceso por fila principal en matrices* (T5), *inicialización frente a asignación* (T15), *bucles con cuenta regresiva* (T19), *desenrollado de bulces* (T20) y *búsqueda lineal* (T25)), el modelo 3B+ (con un diseño más centrado en la eficiencia) obtiene mejores resultados que el 4 (más orientado al rendimiento).

5.2.1.2. Reducción del consumo energético en IoT

Esta sección tiene como objetivo analizar el ahorro energético que puede conseguirse al utilizar cada una de las técnicas propuestas en dispositivos IoT y constatar que, en este caso, la ayuda automática que ofrece el compilador también resulta insuficiente. Así, se evalúa la energía consumida al aplicar estas técnicas y se compara con los resultados conseguidos al utilizar la ayuda ofrecida por el compilador GCC, demostrando que los programadores pueden ahorrar energía si aplican las técnicas propuestas en lugar de emplear únicamente las opciones de optimización automática proporcionadas por el compilador.

5.2.1.2.1. Metodología

Los experimentos desarrollados en los cuatro tipos de dispositivos Raspberry Pi, utilizan en esta ocasión el sistema operativo Raspbian en su versión Buster 4.19, con todos sus paquetes actualizados. Los códigos son compilados con GCC 8.3.0 contenido en la versión estable del paquete "*Raspbian 8.3.0-6+rpi1*" de la distribución. Las tarjetas *microSDHC* utilizadas son las mismas que en los anteriores experimentos (SanDisk Ultra clase 10 con velocidad mínima de escritura de 10 MB/s) y todas las pruebas han sido ejecutadas con una temperatura ambiente de 22 °C, alcanzando las RPi una temperatura máxima de 60 °C. Adicionalmente, también se han realizado pausas de 3 minutos entre las distintas ejecuciones de cada experimento, para evitar alteraciones en los resultados debido a probables sobrecalientamientos durante las pruebas. El resto de detalles específicos relacionados con la medición de los tiempos de ejecución han sido descritos previamente en el apartado 5.2.1.1.1 de este mismo capítulo.

En general, existen dos opciones principales para determinar el consumo energético en un dispositivo electrónico: conectando un instrumento de medición en serie a su circuito de alimentación o empleando una pinza amperimétrica que al abrirse permite sujetar el cable eléctrico a medir. Aunque la utilización de esta última evita las posibles desventajas de añadir un amperímetro a un circuito, su uso está destinado a medir circuitos de alta intensidad.

Como se ha mencionado anteriormente, es posible utilizar el puerto de entrada/salida de propósito general (GPIO) de una Raspberry para conectar los pines de 5 V (voltios) a una fuente de energía y alimentar directamente la placa. Este tipo de alimentación permite conectar un multímetro en serie y realizar mediciones sin necesidad de tener que

modificar el cable (que estaría conectado al puerto micro-USB o USB-C) ni el conector de alimentación de la propia placa.

De este modo, los análisis sobre la energía consumida en cada experimento han sido realizados con un multímetro digital modelo *UNI-T UT61E* (*UNI-T Co.*, Dongguan, China), junto con una fuente de alimentación de corriente continua que suministra una tensión estable de 5 V. El polo positivo de la fuente se conecta al polo positivo del multímetro (cable rojo), mientras que el polo negativo de la propia fuente es conectado al pin número 6 del GPIO[10]. De la misma forma, el polo negativo del multímetro (cable negro) debe conectarse al pin número 2 (o 4), cerrando así el circuito (ver Figura 5.14).

Los resultados son obtenidos con un ordenador mediante la versión 4.01 del *software* de análisis *UT61E Interface Program* proporcionado por el fabricante del multímetro. De esta manera, la intensidad de corriente eléctrica proporcionada por la fuente de alimentación a la Raspberry puede ser monitorizada de forma continua durante las ejecuciones, con una frecuencia de muestreo de dos mediciones por segundo. Así, los valores de intesidad de corriente son obtenidos en tiempo real y transmitidos al ordenador a través de un cable serie y un adaptador USB (*serial-to-usb converter*), siendo almacenados en un fichero de *log* que posteriormente es analizado para extraer los datos y obtener la intensidad de corriente eléctrica media suministrada por la fuente durante el experimento. El multímetro ha sido configurado para medir hasta un máximo de 10 A (amperios), ya que los valores de intensidad instantánea durante todos los experimentos se encuentran en un rango entre 200 mA y 800 mA. En el medidor, este rango implica una resolución de 1 mA con un margen de error de $\pm(0.5\,\% + 10)$, registrando las mediciones en amperios.

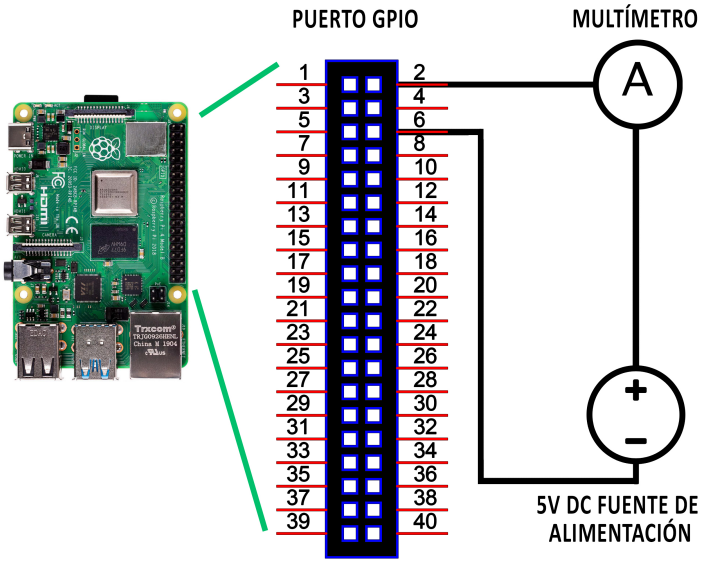

Figura 5.14: Circuito de medición de corriente en dispositivos Raspberry Pi.

[10]Según las especificaciones técnicas de RPi [226], los pines 9, 14, 20, 25, 30, 34 y 39 también serían válidos para el mismo propósito.

La potencia eléctrica (medida en vatios, W) se calcula multiplicando el voltaje (5 V) por la corriente eléctrica, modelada mediante la siguiente ecuación:

$$P_{ower}(W) = V \times I, \tag{5.23}$$

siendo V el voltaje (en voltios, V), I la intensidad de corriente eléctrica (intensidad media durante la ejecución del código, en amperios, A) y P_{ower} la potencia media durante la ejecución del código. La energía (medida en julios) es obtenida mediante la siguiente ecuación:

$$E_{nergy}(J) = P \times t \tag{5.24}$$

siendo P la potencia eléctrica (potencia media durante la ejecución del código, en vatios, W) y t el tiempo (en segundos, s).

5.2.1.2.2. Resultados

Las Tablas 5.9, 5.10, 5.11 y 5.12 muestran el ahorro energético conseguido gracias a la aplicación de las técnicas propuestas en los diferentes dispositivos RPi 2B, 3B, 3B+ y 4. En concreto, proporcionan información correspondiente a la energía consumida en nanojulios por cada código estándar y eficiente, aplicando o no la optimización automática del compilador, así como el porcentaje de ahorro entre las ejecuciones de ambos códigos.

En este caso las desviaciones estándar obtenidas para cuantificar la dispersión de las mediciones han resultado mínimas, con una desviación media del 0,28 % considerando todos los niveles de optimización. Asimismo, al igual que sucede con los datos relativos a los tiempos de ejecución, se omiten las tablas correspondientes a los niveles 1 y 2, dado que sus valores no mejoran los resultados y no son particularmente representativos para el análisis del consumo. No obstante, los resultados completos de los experimentos realizados pueden ser consultados en los materiales suplementarios del artículo publicado sobre esta investigación [238].

Con objeto de mejorar la comprensión de los datos, se ofrecen las Figuras 5.15 y 5.16 que resumen el contenido de las tablas. Estas gráficas muestran en sus ejes horizontales los números que identifican cada una de las técnicas aplicadas, mientras que los porcentajes de ahorro energético alcanzados (respecto a las versiones de código estándar) son ofrecidos en los ejes verticales. Se representan igualmente los cuatro modelos de RPi utilizados en los experimentos: 2B (barras amarillas), 3B (barras rojas), 3B+ (barras azules) y 4B (barras verdes). La Figura 5.15 muestra los resultados cuando se utiliza el nivel de optimización 0, es decir, no se utiliza la optimización automática del compilador, mientras que la Figura 5.16 representa los datos obtenidos cuando los tests son compilados con el tercer nivel de optimización.

En términos de ahorro energético, los porcentajes de mejora al aplicar las técnicas son similares a los alcanzados en términos de disminución del tiempo de ejecución. De igual modo, los resultados relativos al nivel de optimización 0 también deben ser tenidos en cuenta, puesto que, como se ha mencionado anteriormente, en este nivel la capacidad de depuración del código permanece inalterada. Esto pone de manifiesto la importancia que presentan estas técnicas, también en relación a la eficiencia energética, en aquellos casos

Tabla 5.9: Consumo energético y porcentajes de mejora en RPi 2B.

	Técnicas	Energía consumida (sin optimización)			Energía consumida (con optimización nivel 3)		
		Estándar (nJ)	Eficiente (nJ)	Mejora (%)	Estándar (nJ)	Eficiente (nJ)	Mejora (%)
1	Campos de bits	121,96	66,07	45,82	51,22	48,54	5,24
2	Conjuntos de bits	2.529,68	276,74	89,06	2.520,35	276,88	89,01
3	Retorno de booleanos	56,72	52,39	7,62	28,04	27,99	0,18
4	Llamadas en cascada a funciones	1.121,10	573,22	48,87	705,75	116,99	83,42
5	Acceso por fila principal en matrices	13.890,62	13.943,61	0,00	1.951,87	522,36	73,24
6	Listas de inicialización	1.487,91	1.462,10	1,73	1.418,83	1.353,56	4,60
7	Eliminación de subexpresiones	98,03	74,06	24,46	48,72	47,12	3,28
8	Mapeo de estructuras	732,29	1.117,32	0,00	638,66	627,11	1,81
9	Eliminacion de código muerto	44,74	34,12	23,74	23,64	23,53	0,44
10	Control de excepciones	13.250,96	58,27	99,56	12.868,40	14,13	99,89
11	Variables globales en bucles	745,19	572,42	23,18	119,66	121,63	0,00
12	Funciones inline o insertadas	77,21	49,79	35,52	28,02	27,96	0,19
13	Variables globales	2.126,28	1.659,54	21,95	980,14	822,56	16,08
14	Constantes en bucles	673,31	473,71	29,64	315,76	316,25	0,00
15	Inicialización frente a asignación	80,44	39,64	50,73	76,50	13,95	81,76
16	División con potencias de dos	49,98	49,81	0,36	27,98	27,86	0,42
17	Multiplicación con potencias de dos	49,96	49,83	0,25	27,92	27,81	0,37
18	Integer frente a Character	81,54	73,90	9,37	27,82	28,05	0,00
19	Bucles con cuenta regresiva	2.061,13	2.503,40	0,00	511,80	503,34	1,65
20	Desenrollado de bucles	1.058,70	678,36	35,92	121,24	105,82	12,72
21	Paso de estructuras por referencia	572,92	61,22	89,31	547,43	13,89	97,46
22	Aliasing de punteros	164,54	154,48	6,12	99,02	94,18	4,88
23	Cadenas de punteros	112,10	83,87	25,19	34,17	34,06	0,29
24	Pre-incremento vs post-incremento	3.706,04	3.805,12	0,00	991,48	990,06	0,14
25	Busqueda lineal	3.378,41	2.740,79	18,87	969,45	530,29	45,30
26	Estructuras IF en bucles	2.990,67	2.067,03	30,88	185,21	185,20	0,01

Tabla 5.10: Consumo energético y porcentajes de mejora en RPi 3B.

	Técnicas	Energía consumida (sin optimización)			Energía consumida (con optimización nivel 3)		
		Estándar (nJ)	Eficiente (nJ)	Mejora (%)	Estándar (nJ)	Eficiente (nJ)	Mejora (%)
1	Campos de bits	105,17	58,49	44,38	47,40	46,06	2,83
2	Conjuntos de bits	2.080,92	240,81	88,43	2.079,55	240,39	88,44
3	Retorno de booleanos	61,20	47,56	22,30	23,94	24,08	0,00
4	Llamadas en cascada a funciones	998,58	550,24	44,90	592,80	97,03	83,63
5	Acceso por fila principal en matrices	12.681,55	12.696,56	0,00	1.903,67	267,23	85,96
6	Listas de inicialización	1.211,80	1.162,43	4,07	1.191,65	1.150,58	3,45
7	Eliminación de subexpresiones	90,73	69,58	23,31	48,40	46,34	4,26
8	Mapeo de estructuras	754,68	1.022,39	0,00	649,29	715,81	0,00
9	Eliminacion de código muerto	39,63	28,93	27,02	21,20	20,77	2,02
10	Control de excepciones	10.676,17	51,53	99,52	10.861,19	12,86	99,88
11	Variables globales en bucles	703,81	522,70	25,73	147,87	144,62	2,20
12	Funciones inline o insertadas	71,35	45,14	36,73	23,98	24,10	0,00
13	Variables globales	1.743,59	1.449,88	16,84	1.023,41	925,87	9,53
14	Constantes en bucles	635,06	442,52	30,32	281,49	272,23	3,29
15	Inicialización frente a asignación	76,89	36,17	52,96	71,22	12,66	82,23
16	División con potencias de dos	44,38	44,50	0,00	23,99	23,90	0,39
17	Multiplicación con potencias de dos	44,42	44,37	0,11	23,97	23,94	0,11
18	Integer frente a Character	75,89	67,23	11,41	23,83	23,81	0,09
19	Bucles con cuenta regresiva	1.919,36	2402,28	0,00	549,88	393,33	28,47
20	Desenrollado de bucles	993,81	623,35	37,28	94,94	105,22	0,00
21	Paso de estructuras por referencia	592,78	56,22	90,52	498,65	12,76	97,44
22	Aliasing de punteros	153,20	143,67	6,22	90,72	86,33	4,83
23	Cadenas de punteros	108,18	80,63	25,47	31,60	31,50	0,32
24	Pre-incremento vs post-incremento	3.476,96	3.564,70	0,00	1.065,11	1.046,03	1,79
25	Busqueda lineal	3.112,78	2.473,12	20,55	865,26	578,88	33,10
26	Estructuras IF en bucles	2.839,55	1.922,75	32,29	129,04	129,64	0,00

Tabla 5.11: Consumo energético y porcentajes de mejora en RPi 3B+.

	Técnicas	Energía consumida (sin optimización)			Energía consumida (con optimización nivel 3)		
		Estándar (nJ)	Eficiente (nJ)	Mejora (%)	Estándar (nJ)	Eficiente (nJ)	Mejora (%)
1	Campos de bits	149.45	82.73	44.65	66.54	64.53	3.03
2	Conjuntos de bits	2.513,84	290,07	88,46	2.515,70	291,33	88,42
3	Retorno de booleanos	86.61	67.22	22.39	33.68	33.76	0.00
4	Llamadas en cascada a funciones	1.401,71	769,40	45,11	827,05	134,82	83,70
5	Acceso por fila principal en matrices	18.009,01	18.051,36	0,00	2.658,73	375,46	85,88
6	Listas de inicialización	1.689,81	1.617,21	4,30	1.651,05	1.591,84	3,59
7	Eliminación de subexpresiones	128,22	97,78	23,74	67,65	64,60	4,51
8	Mapeo de estructuras	1.068,11	1.421,71	0,00	906,92	995,19	0,00
9	Eliminacion de código muerto	55,53	40,30	27,42	29,71	28,95	2,54
10	Control de excepciones	14.790,17	72,01	99,51	15,265,70	17,88	99,88
11	Variables globales en bucles	987,25	730,92	25,96	205,91	200,43	2,66
12	Funciones inline o insertadas	100,34	63,23	36,98	33,79	33,96	0,00
13	Variables globales	2.427,07	2.012,82	17,07	1.426,08	1.288,63	9,64
14	Constantes en bucles	887,88	617,33	30,47	392,18	379,02	3,36
15	Inicialización frente a asignación	107,27	50,60	52,83	99,65	17,75	82,18
16	División con potencias de dos	62,61	62,64	0,00	33,84	33,75	0,28
17	Multiplicación con potencias de dos	62,92	62,79	0,21	33,87	33,69	0,52
18	Integer frente a Character	106,43	94,60	11,12	33,56	33,64	0,00
19	Bucles con cuenta regresiva	2.683,04	3.353,65	0,00	761,17	541,75	28,83
20	Desenrollado de bucles	1.386,73	868,13	37,40	132,23	147,56	0,00
21	Paso de estructuras por referencia	751,48	79,48	89,42	692,32	17,61	97,46
22	Aliasing de punteros	215,79	201,78	6,49	126,51	120,05	5,11
23	Cadenas de punteros	152,42	113,12	25,79	44,42	44,31	0,25
24	Pre-incremento vs post-incremento	4.843.87	4946.66	0.00	1.474,57	1.442,73	2,16
25	Busqueda lineal	4.346,57	3.483,12	19,87	1.201,38	799,32	33,47
26	Estructuras IF en bucles	3.995,83	2.700,42	32,42	180,29	180,58	0,00

Tabla 5.12: Consumo energético y porcentajes de mejora en RPi 4B.

	Técnicas	Energía consumida (sin optimización)			Energía consumida (con optimización nivel 3)		
		Estándar (nJ)	Eficiente (nJ)	Mejora (%)	Estándar (nJ)	Eficiente (nJ)	Mejora (%)
1	Campos de bits	93,08	40,84	56,12	29,38	29,57	0,00
2	Conjuntos de bits	2.643,58	231,22	91,25	2.640,98	231,18	91,25
3	Retorno de booleanos	43,99	34,30	22,02	18,64	18,65	0,00
4	Llamadas en cascada a funciones	755,31	806,87	0,00	707,72	77,15	89,10
5	Acceso por fila principal en matrices	18.990,96	16.691,18	12,11	2.377,83	516,61	78,27
6	Listas de inicialización	944,10	925,12	2,01	1.430,88	1.282,51	10,37
7	Eliminación de subexpresiones	62,98	59,27	5,88	27,98	31,84	0,00
8	Mapeo de estructuras	880,96	1.133,15	0,00	814,32	855,69	0,00
9	Eliminacion de código muerto	24,48	18,36	24,97	18,72	17,53	6,38
10	Control de excepciones	9.795,13	44,62	99,54	9.510,51	19,93	99,79
11	Variables globales en bucles	486,16	437,48	10,01	131,38	131,05	0,25
12	Funciones inline o insertadas	56,82	32,16	43,41	18,58	18,71	0,00
13	Variables globales	1.300,58	1.163,04	10,58	912,82	822,38	9,91
14	Constantes en bucles	403,26	338,58	16,04	231,32	229,93	0,60
15	Inicialización frente a asignación	54,48	25,63	52,97	53,83	17,78	66,98
16	División con potencias de dos	29,27	32,00	0,00	18,55	19,40	0,00
17	Multiplicación con potencias de dos	29,51	31,98	0,00	18,62	20,54	0,00
18	Integer frente a Character	64,53	58,91	8,71	18,40	18,62	0,00
19	Bucles con cuenta regresiva	3.735,86	3.788,18	0,00	564,20	552,82	2,02
20	Desenrollado de bucles	1.922,48	917,66	52,27	143,33	126,70	11,61
21	Paso de estructuras por referencia	409,63	45,26	88,95	378,96	18,63	95,08
22	Aliasing de punteros	103,04	105,81	0,00	71,62	62,04	13,37
23	Cadenas de punteros	78,29	89,63	0,00	25,91	26,54	0,00
24	Pre-incremento vs post-incremento	7.309,77	7.349,94	0,00	1.077,51	1.068,81	0,81
25	Busqueda lineal	2.382,02	1.684,48	29,28	641,00	591,53	7,72
26	Estructuras IF en bucles	4.031,94	3.760,77	6,73	170,33	170,32	0,00

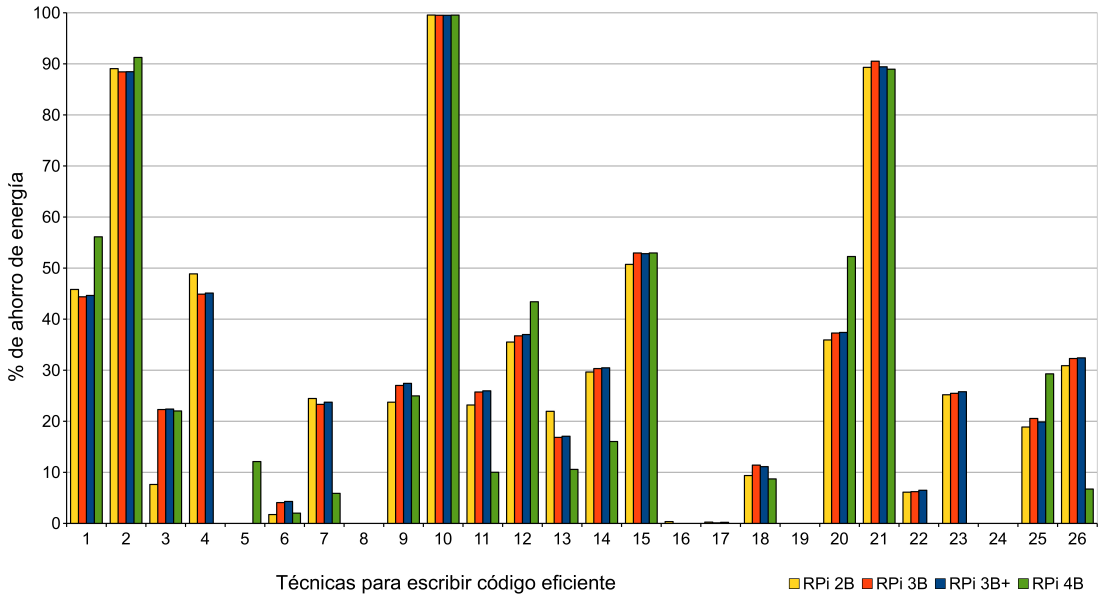

Figura 5.15: Porcentajes de ahorro energético obtenidos mediante la escritura de código eficiente (sin la optimización del compilador) sobre dispositivos IoT.

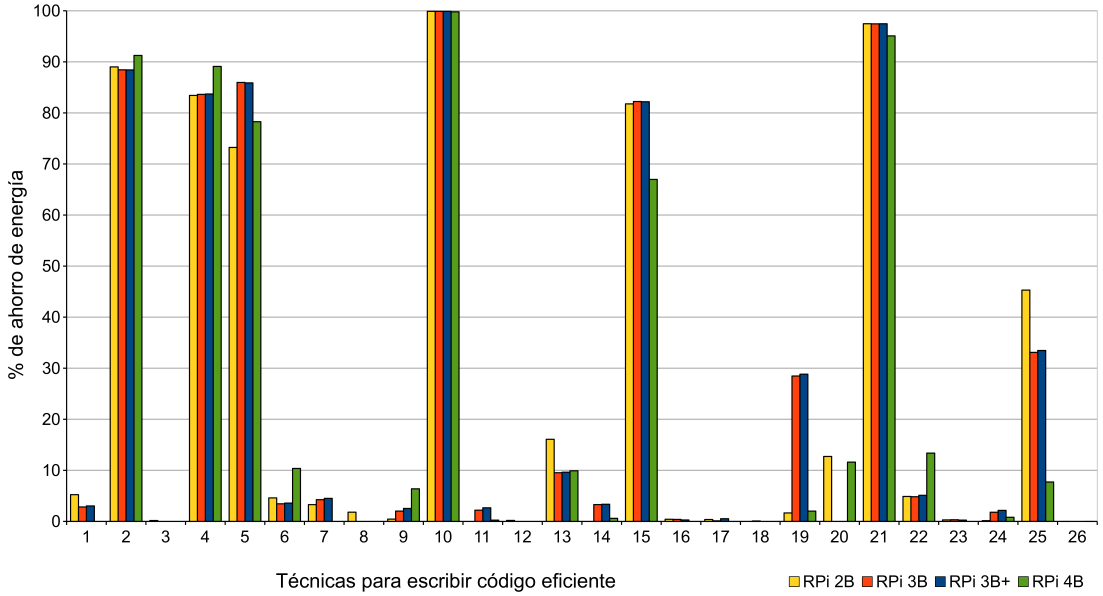

Figura 5.16: Porcentajes de ahorro energético obtenidos mediante la escritura de código eficiente (con nivel 3 de optimización) sobre dispositivos IoT.

que implican considerables procesos de depuración, con los cuales no es posible aplicar las optimizaciones automáticas del compilador (ver Figura 5.15).

Como puede observarse en las figuras y las tablas, el uso de estas técnicas permite alcanzar ahorros energéticos de hasta un 99,89 %. De nuevo, tal y como sucede con los tiempos de ejecución, destacan especialmente las mejoras obtenidas al aplicar las técnicas de *conjuntos de bits* (T2), *manejo de excepciones* (T10) y *paso de estructuras por referencia* (T21).

También de forma análoga a los resultados obtenidos en los tiempos de ejecución, en el caso de los porcentajes de ahorro energético, estos también son similares en los cuatros dispositivos, salvo por ciertas diferencias originadas principalmente por los resultados del modelo RPi 4 que obtiene, en el caso del nivel de optimización 0 (ver Figura 5.15), ahorros de energía superiores hasta en un 16 % al del resto al utilizar: *campos de bits* (T1); *acceso por fila principal en matrices* (T5); *funciones inline o insertadas* (T12); *desenrollado de bucles* (T20); y *búsqueda lineal* (T25). Por el contrario, el porcentaje de mejora energética con este modelo es mucho menor (e incluso inexistente) en los siguientes casos (en comparación con los resultados de los otros tres modelos): *llamadas en cascada a funciones* (T4); *eliminación de subexpresiones* (T7); *variables globales en bucles* (T11); *variables globales* (T13); *constantes en bucles* (T14); *aliasing de punteros* (T22); *cadenas de punteros* (T23); y *estructuras IF en bucles* (T26).

Continuando con la comparación de resultados entre los cuatro modelos RPi, en el nivel de optimización 3 (ver Figura 5.16) el 2B obtiene significativas mejoras, con diferencias de hasta un 37,58 % frente a otros dispositivos al aplicar: *campos de bits* (T1); *uso de variables globales* (T13) y *búsqueda lineal* (T25). En cualquier caso, estas diferencias no implican que sus consumos energéticos sean inferiores, sino que el impacto de aplicar las técnicas es mayor en este modelo. En este nivel la RPi 4 alcanza mejores resultados que otros modelos (hasta un 11,61 % más de ahorro) cuando se aplican: *llamadas en cascada a funciones* (T4); *listas de inicialización* (T6); *eliminación de código muerto* (T9); y *aliasing de punteros* (T22), mientras que su rendimiento es menor con las siguientes: *campos de bits* (T1); *eliminación de subexpresiones* (T7); *variables globales* (T13); *inicialización frente a asignación* (T15); *bucles con cuenta regresiva* (T19); y *búsqueda lineal* (T25).

En general, en el nivel de optimización 0, relativo al ahorro energético obtenido sin la ayuda de las optimizaciones automáticas del compilador (Figura 5.15) y considerando todos los sistemas RPi utilizados, varias de las técnicas propuestas no producen mejoras o bien éstas son muy limitadas. Es decir, con ellas el código eficiente no consume menos energía que el correspondiente código estándar, lo cual sucede cuando se aplican las siguientes: *acceso por fila principal en matrices* (T5), excepto en la RPi 4; *mapeo de estructuras* (T8); *división con denominadores potencias de 2* (T16); *multiplicación con factores potencia de dos* (T17); *bucles con cuenta regresiva* (T19); y *pre-incremento frente a pos-incremento* (T24). Las otras 19 técnicas demuestran su eficiencia en la escritura de código cuando la optimización del compilador está configurada por defecto (es decir, con nivel 0), logrando un ahorro medio de energía del 35,25 % considerando los cuatro modelos diferentes de RPi.

Cuando se utiliza el nivel 3 de optimización y además se aplican las técnicas, la ayuda automática ofrecida por el compilador alcanza resultados destacables en la mayoría de las pruebas realizadas (Figura 5.16). Sin embargo, la escritura manual de código eficiente por parte del programador sigue siendo recomendable respecto a las siguientes técnicas, que logran un ahorro medio de energía del 71,90 %: *conjuntos de bits* (T2), *llamadas en cascada a funciones* (T4); *acceso por fila principal en matrices* (T5); *control de excepciones* (T10); *inicialización frente a asignación* (T15); *bucles con cuenta regresiva* (T19); *paso de estructuras por referencia* (T21); y *búsqueda lineal* (T25).

A continuación estas ocho técnicas son analizadas con mayor profundidad utilizando para ello los modelos RPi 3B+ y 4. Asimismo, cabe destacar que se trata de las mismas que obtuvieron mejor rendimiento en relación a los tiempos de ejecución, a excepción del *desenrollado de bucles* (T20) que, sin embargo, no alcanza porcentajes significativos de ahorro energético.

Conjuntos de bits (T2)

De forma análoga a lo sucedido en los experimentos relativos a los tiempos de ejecución, esta técnica también posibilita importantes disminuciones en el consumo energético al utilizar conjuntos de *bits* en lugar de vectores (ver el resumen de ambos tests en el Código 5.4 en la página 89 o los tests completos en el apartado C.2 del Apéndice correspondiente). La Figura 5.17 muestra gráficamente cómo varia el ahorro energético en función del número de elementos utilizados.

Las ventajas de emplear los conjuntos de *bits* comienzan a evidenciarse a partir de un número superior a 5 booleanos. Con 10 se obtiene una reducción de consumo entre el 38 % (RPi 3B+) y el 50 % (RPi 4B), con 30 elementos la mejora ya se sitúa entre el 76 y el 85 % (3B+ y 4B respectivamente), mientras que con 100 elementos el ahorro crece hasta el 90 %, estabilizándose en torno al 94 % en ambos dispositivos a partir de 250 elementos.

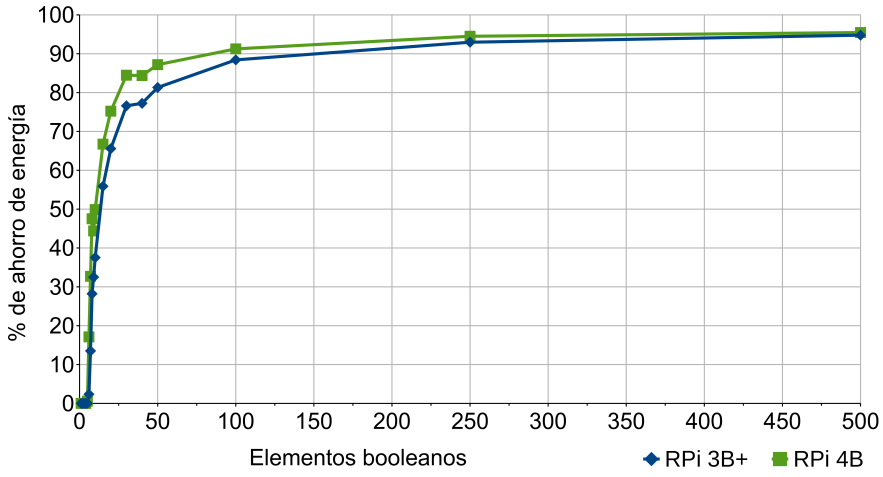

Figura 5.17: Conjuntos de bits (T2). Porcentajes de ahorro energético según el número de elementos booleanos requeridos en RPi 3B+ y 4B.

Llamadas en cascada a funciones (T4)

Los experimentos realizados demuestran que también es recomendable aplicar esta técnica en relación a la eficiencia energética, obteniendo resultados significativamente similares a los relacionados con el tiempo de ejecución. La Figura 5.18 muestra cómo varía la reducción del consumo según el número de llamadas a la función, de forma que gracias a la utilización de esta propuesta se obtienen ahorros de hasta el 95 %, alcanzando una mejora del 89,04 % cuando se utiliza un bucle de 20 iteraciones, es decir, con tan solo 20 llamadas a la función (ver Código 5.5 -pág. 90- o Test C.4). Asimismo, son destacables las mejoras del 45,71 % con dos únicas llamadas y del 52,31 % realizando únicamente tres. A partir de 100 o más, el ahorro se estabiliza en torno al 96 %, demostrándose así la eficacia de la técnica. Ambos dispositivos obtienen resultados análogos, con rendimiento superior en RPi 4 cuando se realizan entre 2 y 50 llamadas.

Acceso por fila principal en matrices (T5)

Recorrer las matrices por fila principal siguiendo el orden en que los elementos son almacenados (ver Código 5.6 -pág. 91- o Test C.5), obtiene unos porcentajes de ahorro energético que crecen a medida que aumenta el tamaño de la matriz, logrando hasta un 85,88 % en el dispositivo 3B y 78,27 % en el modelo 4 (porcentajes similares a los obtenidos en relación a los tiempos de ejecución). Estas mejoras tienden a estabilizarse alrededor del 80 % en la RPi 3B y el 60 % en la placa 4B cuando se emplean matrices con un orden igual o superior a 40 (ver Figura 5.19), tal y como sucede con las disminuciones de los tiempos de ejecución. Consecuentemente, en este caso también se analizarán en mayor profundidad los efectos producidos por las variaciones en las dimensiones de las matrices al aplicar esta técnica. Asimismo, las conclusiones sobre las diferencias existentes entre los resultados obtenidos por ambos modelos en relación al tiempo de ejecución, pueden ser aplicadas igualmente respecto al ahorro energético.

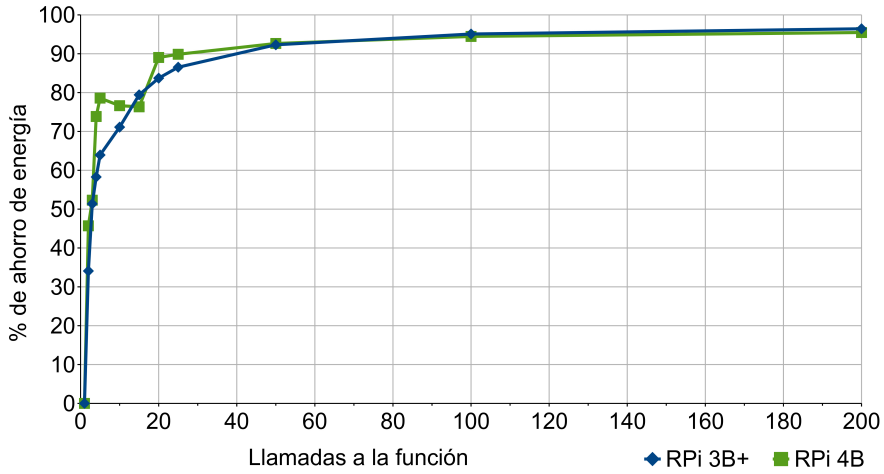

Figura 5.18: Llamadas en cascada a funciones (T4). Porcentajes de ahorro energético según el número de llamadas a la función (número de iteraciones del bucle) en RPi 3B+ y 4.

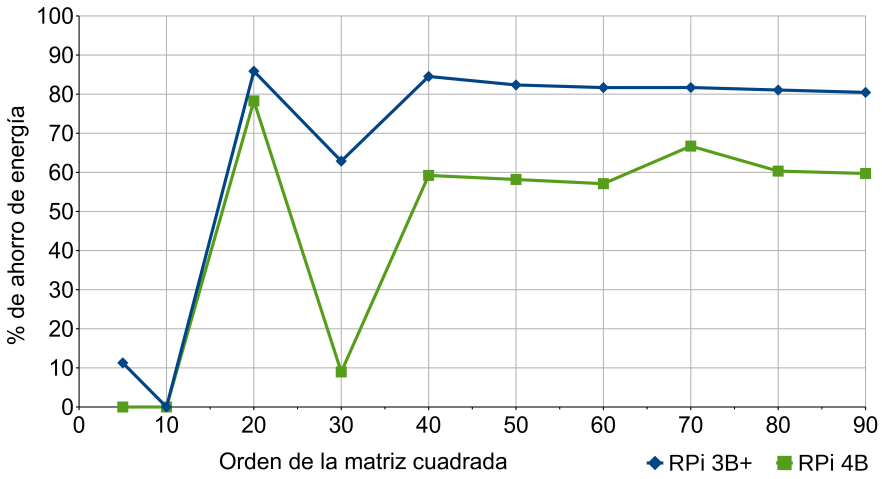

Figura 5.19: Acceso por fila principal en matrices (T5). Porcentajes de ahorro energético en función del orden de la matriz cuadrada en RPi 3B+ y 4.

Control de excepciones (T10)

Los experimentos realizados demuestran que evitar el lanzamiento reiterado de excepciones dentro de bucles mediante la utilización de sentencias como "`continue`" y "`break`" (ver Código 5.7 -pág. 92- o Test C.10), no solamente evita penalizaciones de tiempo sino que además implica notables ahorros de energía. Por tanto, es posible cambiar los flujos de ejecución y controlar los posibles errores de forma semejante a la de las excepciones, pero con un coste menor.

En este caso también pueden realizarse numerosas variaciones para analizar este experimento, dado que el porcentaje de mejora energética depende del número de excepciones lanzadas y de la estructura de los datos. Por ello, la evaluación en mayor profundidad de las condiciones que influyen en el ahorro, será abordada en posteriores investigaciones, puesto que además las diferencias obtenidas entre los diferentes modelos de RPi no son significativas (ver Figura 5.16 -pág. 103-).

En la versión eficiente la omisión del lanzamiento de las 100 excepciones simples del código estándar obtiene un ahorro energético del 99 % pese a que, como se mencionaba con anterioridad, estas excepciones no presenten ningún contenido aparte de su declaración. Por ello se recomienda aplicar esta técnica bajo las condiciones indicadas para mejorar la eficiencia.

Inicialización frente a asignación (T15)

La Figura 5.16 (pág. 103) muestra de forma gráfica cómo la técnica relativa a inicializar directamente las variables al ser declaradas (ver Código 5.8 -pág. 93- o Test C.15) consigue ahorros de energía de hasta un 52,97 % (un porcentaje significativamente inferior al 80 % que se alcanza con esta misma técnica en el tiempo de ejecución). La mejora depende de los tipos de datos, puesto que existen diversas variaciones posibles que pueden ser aplicadas y analizadas en este experimento (considerando tipos de datos primitivos, derivados y abstractos o definidos por el usuario, así como modificadores de tipos de datos, como por ejemplo, *unsigned* o *long*). Por ello, del mismo modo que en la técnica anterior, en

este caso también se ha omitido la evaluación en mayor profundidad de las condiciones que influyen en los porcentajes de mejora, así como el gráfico comparativo, ya que las diferencias obtenidas entre los diferentes modelos de RPi tampoco son relevantes (ver nuevamente Figura 5.16).

Bucles con cuenta regresiva (T19)

En la Figura 5.20 se observa cómo al aplicar esta técnica, centrada en recorrer los bucles en sentido opuesto utilizando estructuras `while` en lugar de `for` (ver Código 5.9 -pág. 93- o Test C.19), se logran ahorros energéticos de entre el 20 % y el 30 % utilizando el modelo 3B+ y vectores con más de 20 elementos, mientras que en el modelo 4 la mejora es prácticamente nula (entre un 1,29 % y un 2,86 %). Estos porcentajes son notablemente similares a los conseguidos en la reducción de tiempos de ejecución, por los mismos motivos debatidos en el apartado correspondiente.

Paso de estructuras por referencia (T21)

El código eficiente de esta técnica, que pasa la clase por referencia y accede al índice de la subestructura a través de un puntero (ver Código 5.11 -pág. 96- o Test C.21), alcanza una mejora energética de alrededor del 90,54 % en los cuatro modelos RPi, frente al 97,95 % que alcanzaba en la disminución del tiempo de ejecución.

De igual modo que en otras técnicas, el porcentaje de ahorro energético ofrecido por ésta depende de las estructuras de datos utilizadas, por lo que existen infinidad de posibles variaciones que pueden ser aplicadas y analizadas en este experimento. Por ello, en este caso también se omiten tanto el gráfico comparativo como la evaluación en mayor profundidad de las condiciones que influyen en estas mejoras, ya que además, las diferencias obtenidas entre los distintos modelos de RPi no resultan destacables. En cualquier caso, todas estas cuestiones serán tratadas en futuros trabajos.

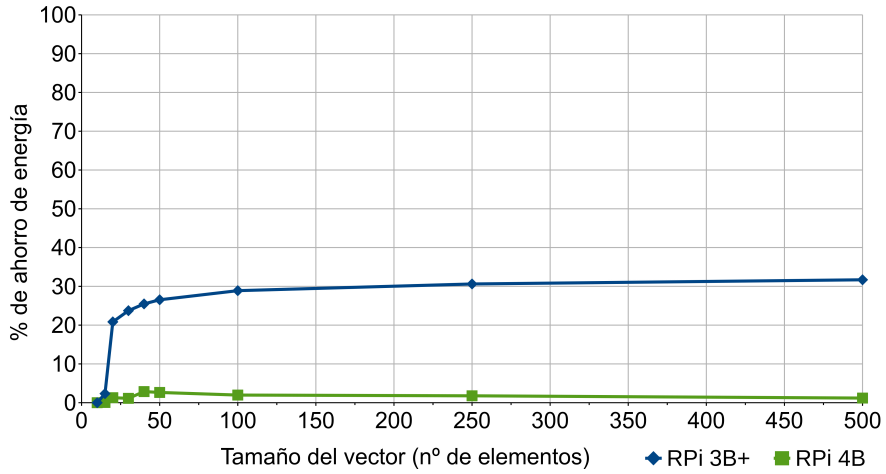

Figura 5.20: Bucles con cuenta regresiva (T19). Porcentajes de ahorro energético según el tamaño del vector (número de elementos) en RPi 3B+ y 4.

Búsqueda lineal (T25)

En la Figura 5.21 puede observarse que al aplicar esta técnica en la RPi 3B+ utilizando vectores de entre 20 y 30 elementos, se obtienen mejoras de entre un 20 % y un 30 %, mientras que con el modelo 4 el porcentaje se reduce considerablemente y únicamente alcanza el 7,63 %. Estas diferencias entre ambos dispositivos en términos de energía son similares a las analizadas en los experimentos de reducción del tiempo de ejecución, siendo también debidas, tal y como se describe en el correspondiente apartado, al escaso margen de mejora existente en los resultados correspondientes a la RPi 4.

De este modo, se ha demostrado la efectividad de las técnicas propuestas para escribir código eficiente, consiguiendo significativas reducciones sobre el tiempo de ejecución y el consumo energético, con porcentajes de mejora similares en ambos parámetros. Estas mejoras son análogas en los cuatro modelos de Raspberry Pi utilizados. Sin embargo, en ciertos experimentos el modelo 3B+ (con Cortex A-53) obtiene mayores ahorros que el modelo 4 (Cortex-A72), debido a las diferencias entre ambos: el primero, centrado en la eficiencia y el segundo, orientado al rendimiento. Adicionalmente, las técnicas que obtienen mejores resultados han sido analizadas con mayor profundidad.

Todos los datos obtenidos mediante la aplicación manual de las técnicas han sido comparados con los logrados al aplicar directamente las optimizaciones automáticas ofrecidas por el compilador GCC. Así, se ha demostrado también que el usuario desempeña un papel fundamental para aumentar el rendimiento, puesto que el compilador no consigue las mismas mejoras que pueden alcanzar los programadores al aplicar las técnicas de forma manual. Por lo tanto, estos deben ser conscientes del importante impacto que cualquier instrucción puede tener en el rendimiento final de su código. Además, los resultados obtenidos en la investigación desarrollada pueden ser extrapolados a otros dispositivos con arquitecturas ARM similares, especialmente aquellos relacionados con IoT.

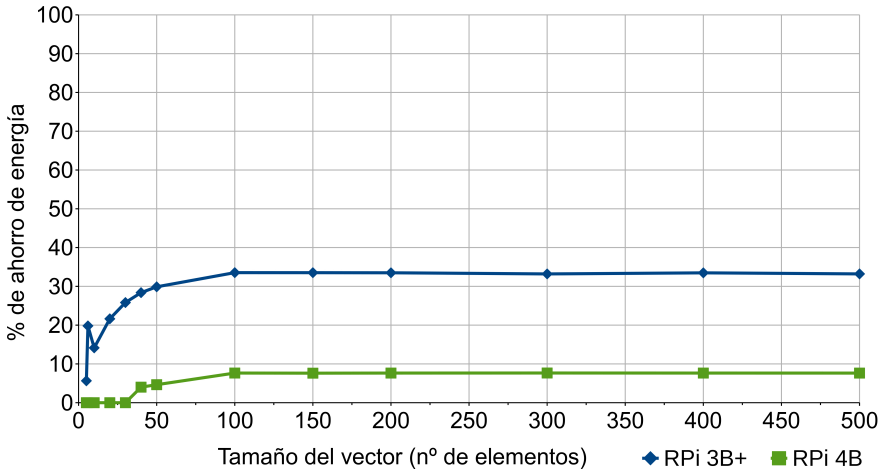

Figura 5.21: Busqueda lineal (T25). Porcentajes de ahorro energético según el tamaño del vector (número de elementos) en RPi 3B+ y 4.

5.2.2. Aplicación de las técnicas en infraestructuras HPC

En este apartado se recogen los resultados extraídos al analizar la aplicación de las técnicas propuestas sobre infraestructuras de cómputo de alto rendimiento. Los distintos experimentos han sido centrados en la reducción de los tiempos de ejecución, con la intención de evaluar en próximas investigaciones los ahorros energéticos alcanzados mediante la aplicación de estas técnicas. Para ello se considerará la utilización de simuladores que permiten reproducir el comportamiento en infraestructuras HPC y se emplearán *benchmarks* para estudiar el consumo energético [243, 244].

5.2.2.1. Reducción del tiempo de ejecución en HPC

A continuación se detalla la metodología seguida y se discuten los resultados alcanzados en relación a la reducción de los tiempos de ejecución en sistemas de cómputo de alto rendimiento, publicados en [240].

5.2.2.1.1. Metodología

Para desarrollar los experimentos se han utilizado tres nodos de cómputo: el servidor Fujitsu Primergy RX4770 M2 con 4 procesadores Intel Xeon E7-4830v3 correspondiente al nodo de cómputo de memoria compartida (descrito previamente en el apartado 5.1.1 relativo a los experimentos sobre la paralelización automática de códigos), el servidor Fujitsu Primergy CX2550 con 2 procesadores Intel Xeon E5-2660v3 de 10 núcleos y 20 cores en total (detallado también con anterioridad) y un tercer servidor IBM System x iDataPlex dx360 M4 con 2 procesadores Intel Xeon E5-2670 de 8 núcleos y 16 cores en total. Estos nodos presentan unas especificaciones principales que pueden ser consultadas en la Tabla 5.13. Fueron seleccionados por su representatividad en lo que a *fat nodes* y clústeres de memoria distribuida se refiere, así como por ser ampliamente utilizados en centros HPC, incluyendo diez de ellos [245–252] clasificados entre los sistemas de cómputo más potentes del mundo [50]. Además, estos nodos son frecuentemente utilizados por investigadores y usuarios del centro de supercomputación donde se ha desarrollado esta tesis doctoral [15].

En cualquier caso, la efectividad y mejoras logradas gracias a la aplicación de las técnicas analizadas en este apartado serían semejantes en otro tipo de máquinas. De hecho, en futuros trabajos se espera ampliar la presente investigación con el fin de incluir más infraestructuras y arquitecturas de cómputo, considerando nodos de memoria compartida y clústeres de memoria distribuida, con objeto de demostrar que los resultados obtenidos pueden ser extrapolados a un mayor número de centros HPC.

El procesador E5-2670 es similar al E5-2660v3, con mismo *socket* y arquitectura, aunque presentan ciertas diferencias entre ellos. El E5-2660v3 pertenece a una microarquitectura Haswell y tiene dos núcleos más de CPU, 20, frente a los 16 que ofrece el procesador E5-2670, con microarquitectura Sandy Bridge. Ambos cuentan con cachés inteligentes (*smart caches*) de 25 y 20 MB, con velocidades de bus de 9,6 GT/s y 8 GT/s

Tabla 5.13: Especificaciones de los nodos de cómputo donde se han desarrollado los experimentos de las técnicas de optimización, según su procesador.

	Intel Xeon E7-4830v3	**Intel Xeon E5-2660v3**	**Intel Xeon E5-2670**
Nodo de cómputo	Memoria compartida	Memoria distribuida	Memoria distribuida
Nº de procesadores	4	2	2
Nombre clave	Haswell	Haswell	Sandy Bridge
Nº de Cores	12 (por procesador)	10 (por procesador)	8 (por procesador)
Nº total de Cores	48 (por nodo)	20 (por nodo)	16 (por nodo)
Frecuencia base	2,10 GHz	2,60 GHz	2,60 GHz
Caché	30 MB LLC	25 MB SmartCache	20 MB SmartCache
Memoria	1,5 TB DDR4 @ 133 MHz	16 GB DDR4 @ 2133 MHz por core (80 GB por nodo)	2 GB DDR3 @ 1600 MHz por core (32 GB por nodo)
Disco duro local	Fujitsu PRAID CP400i SAS 300 GB 12 Gbps	Innodisk SATADOM-MH 3ME 120 GB 6 Gbps	IBM NL SATA 3.5 7.2K 500 GB 6 Gbps

respectivamente. Además, el nodo de cómputo del E5-2660v3 incluye un ancho de banda de memoria DDR4 más rápido y con 48 GB más (80GB de DDR4 a 2133 MHz frente a los 32 GB de DDR3 a 1600 MHz del E5-2670), que es especialmente útil para tareas limitadas que requieren un uso intensivo de la memoria. Asimismo, los discos duros locales también presentan significativas diferencias: una unidad de estado sólido (*solid-state drive*) y una unidad de disco duro (*hard disk drive*) respectivamente, con todo lo que esto implica en términos de rendimiento.

Aunque los resultados analizados pueden ser extrapolados a otros sistemas operativos, todas las cargas de trabajo utilizadas en los tres nodos de cómputo han sido ejecutadas bajo *SUSE Linux Enterprise Server 12 SP3*. Las pruebas desarrolladas en cada uno de los experimentos realizados en el presente estudio fueron compiladas con GCC v4.8.5.

El algoritmo de medición aplicado en cada test desarrollado es el mismo seguido para el análisis de las técnicas en dispositivos IoT (ver Algoritmo 6 -pág. 82-). De igual modo, por tanto, cada test es repetido cien millones de veces para cada medición con el fin de garantizar que el tiempo de ejecución alcance valores medibles y que el coste de invocación de la función de cada test pueda ser descartado, asegurando la precisión de los resultados. El impacto de los arranques en frío y los efectos de la memoria caché son reducidos realizando diez mediciones distintas (de cien millones de repeticiones cada una). De esta forma el estado de las cachés tiende a converger a un único valor, anulando las variaciones aleatorias y evitando los valores atípicos (*outliers*). Posteriormente se calcula la media de las diez mediciones y se divide por el número de repeticiones de cada test, obteniendo así el resultado final.

Los tiempos de ejecución son medidos también de igual forma que en los dispositivos IoT, mediante la clase *Stopwatch* (ver Código 5.3 -pág. 83-), basada en el temporizador de la librería *Chrono* que, como se mencionaba en la metodología descrita en el apartado 5.2.1.1, dispone de una resolución de 10 milisegundos y presenta un *overhead* que no resulta significativo cuando se cronometran tareas de decenas de milisegundos o más, como es el caso.

Los tests son enviados como trabajos de cómputo mediante *batch scripts* a *Slurm*, el gestor de colas utilizado en el centro de supercomputación, para acceder a los nodos (ver apartado 2.1.4 decicado a Slurm en el Capítulo 2). Con el objetivo de evitar variaciones y alteraciones de los resultados de los experimentos, los scripts por lotes invocan el comando *sbatch* de *Slurm* con la opción "`--exclusive`" que asegura que la asignación de trabajos no pueda compartir nodos con otros trabajos en ejecución.

5.2.2.1.2. Resultados

Las Tablas 5.14, 5.15 y 5.16 muestran los resultados de aplicar las distintas técnicas en los tres nodos de cómputo utilizados (con procesadores Intel Xeon E7-4830v3, E5-2660v3 y E5-2670). Aunque nuevamente se tuvieron en cuenta los cuatro niveles de optimización automática proporcionados por el compilador (ver metodología seguida en el apartado 5.2.1.1 dedicado a la reducción de tiempos de ejecución en IoT), las tres tablas muestran únicamente los resultados de aplicar los dos niveles más representativos: 3 (nivel máximo de optimización) y 0 (sin optimización), ya que los niveles intermedios (1 y 2) no consiguen mejoras relevantes en los resultados.

Al igual que en el caso de los experimentos relativos a los dispositivos IoT, la columna *Estándar* muestra para cada técnica los tiempos de ejecución del primer test, que contiene el código estándar, mientras que la columna *Eficiente* presenta los resultados de aplicar la técnica correspondiente sobre el código estándar con objeto de reducir su tiempo de ejecución. Los datos de ambas columnas son ofrecidos en nanosegundos. Adicionalmente, los porcentajes de mejora logrados gracias a la programación de código eficiente, figuran en la columna *Mejora* para los dos niveles de optimización. La desviación media obtenida es del 0,92 %, considerando los cuatro niveles del compilador (es decir, incluyendo también los no mostrados en las tablas) y confirmando así la precisión de las medidas y la proximidad entre todos los valores obtenidos y los promedios representados.

Cabe mencionar nuevamente que cada resultado de tiempo corresponde a una única ejecución del código y por tanto, a pesar de que en ocasiones estos puedan parecer insignificantes, su importancia crece sustancialmente cuando estos códigos son ejecutados repetidamente en bucles con un gran número de repeticiones, y más aún respecto a la realización de trabajos de HPC que requieren días o incluso semanas de ejecución o necesitan miles de horas de CPU anuales, algo muy común en este tipo de centros tecnológicos. Además, es importante señalar que las técnicas propuestas no son excluyentes entre sí y varias de ellas podrían ser aplicadas en el mismo fragmento de código.

Los tiempos son más reducidos en los experimentos realizados en el nodo de cómputo con el procesador E5-2660v3, debido a las diferencias de rendimiento entre los tres procesadores, descritas anteriormente. Además, en el nivel máximo de optimización (3) se producen disminuciones de tiempo más destacables en comparación con los otros niveles, especialmente cuando se aplican las técnicas evaluadas. Sin embargo, como se ha detallado con anterioridad, también es fundamental analizar los efectos de escribir código eficiente sin la ayuda del compilador, ya que el nivel 0, a diferencia del resto, permite depurar los programas. Así, se demuestra por tanto que también en el caso del HPC, el uso de estas

Tabla 5.14: Tiempos de ejecución y porcentajes de mejora en Intel Xeon E7-4830v3.

	Técnicas	Tiempos de ejecución (sin optimización)			Tiempos de ejecución (optimización nivel 3)		
		Estándar (ns)	Eficiente (ns)	Mejora (%)	Estándar (ns)	Eficiente (ns)	Mejora (%)
1	Campos de bits	5,84	3,82	34,59	4,16	3,71	10,82
2	Conjuntos de bits	105,32	23,27	77,91	105,33	23,80	77,40
3	Retorno de booleanos	4,46	3,71	16,82	2,35	2,32	1,28
4	Llamadas en cascada a funciones	64,79	46,33	28,49	40,21	12,27	69,49
5	Acceso por fila principal en matrices	9.471,62	10.903,65	0,00	522,49	1.086,51	0,00
6	Listas de inicialización	136,17	127,61	6,29	133,58	126,86	5,03
7	Eliminación de subexpresiones	6,32	3,25	48,58	3,06	2,84	7,19
8	Mapeo de estructuras	80,61	107,18	0,00	182,00	70,77	61,12
9	Eliminación de código muerto	2,65	2,34	11,70	2,35	2,34	0,43
10	Control de excepciones	1.563,40	4,08	99,74	1.565,60	2,28	99,85
11	Variables globales en bucles	49,79	51,41	0,00	2,83	2,67	5,65
12	Funciones inline o insertadas	5,80	3,34	42,41	2,34	2,34	0,00
13	Variables globales	149,06	159,24	0,00	113,45	112,15	1,15
14	Constantes en bucles	43,48	31,23	28,17	19,51	21,20	0,00
15	Inicialización frente a asignación	7,62	2,48	67,45	6,59	2,28	65,40
16	División con potencias de dos	3,34	3,71	0,00	2,35	2,34	0,43
17	Multiplicación con potencias de dos	3,35	3,72	0,00	2,34	2,34	0,00
18	Integer frente a Character	4,08	4,08	0,00	2,35	2,34	0,43
19	Bucles con cuenta regresiva	290,93	287,20	1,28	25,25	56,18	0,00
20	Desenrollado de bucles	145,30	78,36	46,07	16,76	9,29	44,57
21	Paso de estructuras por referencia	25,97	5,28	79,67	33,36	2,29	93,14
22	Aliasing de punteros	8,31	6,80	18,17	4,79	4,24	11,48
23	Cadenas de punteros	17,07	15,02	12,01	3,72	3,74	0,00
24	Pre-incremento vs post-incremento	568,24	547,32	3,68	45,13	45,54	0,00
25	Búsqueda lineal	317,92	286,04	10,03	65,14	54,62	16,15
26	Estructuras IF en bucles	255,39	281,89	0,00	26,67	26,37	1,12

Tabla 5.15: Tiempos de ejecución y porcentajes de mejora en Intel Xeon E5-2660v3.

	Técnicas	Tiempos de ejecución (sin optimización)			Tiempos de ejecución (optimización nivel 3)		
		Estándar (ns)	Eficiente (ns)	Mejora (%)	Estándar (ns)	Eficiente (ns)	Mejora (%)
1	Campos de bits	4,78	3,32	30,54	3,41	3,03	11,14
2	Conjuntos de bits	86,02	19,44	77,40	86,01	19,05	77,85
3	Retorno de booleanos	3,65	3,03	16,99	1,92	1,9	1,04
4	Llamadas en cascada a funciones	52,98	37,86	28,54	32,85	10,02	69,50
5	Acceso por fila principal en matrices	7.747,7	8.850,86	0,00	428,92	886,94	0,00
6	Listas de inicialización	111,21	105,2	5,40	109,15	103,42	5,25
7	Eliminación de subexpresiones	5,17	2,66	48,55	2,53	2,3	9,09
8	Mapeo de estructuras	65,83	70,34	0,00	147,92	55,02	62,80
9	Eliminacion de código muerto	2,18	1,91	12,39	1,95	1,91	2,05
10	Control de excepciones	1.276,84	3,34	99,74	1.274,98	1,88	99,85
11	Variables globales en bucles	40,68	41,75	0,00	2,32	2,18	6,03
12	Funciones inline o insertadas	4,74	2,73	42,41	1,94	1,91	1,55
13	Variables globales	121,85	129,95	0,00	92,49	91,91	0,63
14	Constantes en bucles	40,97	29,21	28,70	15,93	17,33	0,00
15	Inicialización frente a asignación	6,18	2,03	67,15	5,46	1,87	65,75
16	División con potencias de dos	2,73	3,03	0,00	1,93	1,91	1,04
17	Multiplicación con potencias de dos	2,75	3,03	0,00	1,92	1,91	0,52
18	Integer frente a Character	3,37	3,34	0,89	1,92	1,91	0,52
19	Bucles con cuenta regresiva	237,7	234,31	1,43	18,25	45,88	0,00
20	Desenrollado de bucles	118,74	63,7	46,35	13,7	8,2	40,15
21	Paso de estructuras por referencia	22,13	4,31	80,52	28,63	1,9	93,36
22	Aliasing de punteros	6,82	5,56	18,48	4,17	3,46	17,03
23	Cadenas de punteros	14,06	12,44	11,52	3,07	3,03	1,30
24	Pre-incremento vs post-incremento	466,08	446,91	4,11	36,91	37,56	0,00
25	Busqueda lineal	259,54	233,4	10,07	53,2	44,56	16,24
26	Estructuras IF en bucles	208,56	230,08	0,00	21,79	21,51	1,28

Tabla 5.16: Tiempos de ejecución y porcentajes de mejora en Intel Xeon E5-2670.

	Técnicas	Tiempos de ejecución (sin optimización)			Tiempos de ejecución (optimización nivel 3)		
		Estándar (ns)	Eficiente (ns)	Mejora (%)	Estándar (ns)	Eficiente (ns)	Mejora (%)
1	Campos de bits	5,8	3,63	37,41	4,44	3,63	18,24
2	Conjuntos de bits	84,25	24,41	71,03	79,11	24,3	69,28
3	Retorno de booleanos	5,24	3,63	30,73	2,17	2,12	2,30
4	Llamadas en cascada a funciones	89,04	52,17	41,41	44,95	10,7	76,20
5	Acceso por fila principal en matrices	9.276,92	9.435,42	0,00	510,29	981,36	0,00
6	Listas de inicialización	139,51	132,15	5,28	140,25	148,11	0,00
7	Eliminación de subexpresiones	5,21	2,96	43,19	2,77	2,67	3,61
8	Mapeo de estructuras	67,12	87,19	0,00	155,77	67,14	56,90
9	Eliminacion de código muerto	2,49	2,12	14,86	2,17	2,12	2,30
10	Control de excepciones	1.583,2	4,24	99,73	1.573,98	2,32	99,85
11	Variables globales en bucles	44	44,48	0,00	2,83	2,74	3,18
12	Funciones inline o insertadas	5,9	3,33	43,56	2,17	2,12	2,30
13	Variables globales	122,09	132,63	0,00	97,09	97,86	0,00
14	Constantes en bucles	45,22	31,65	30,01	19,79	22,73	0,00
15	Inicialización frente a asignación	7,46	2,45	67,16	7,88	2,41	69,42
16	División con potencias de dos	3,1	3,94	0,00	2,14	2,12	0,93
17	Multiplicación con potencias de dos	3,1	3,94	0,00	2,16	2,12	1,85
18	Integer frente a Character	4,31	4,04	6,26	2,18	2,07	5,05
19	Bucles con cuenta regresiva	240,21	239,09	0,47	30,76	55,18	0,00
20	Desenrollado de bucles	118,66	64,17	45,92	15,53	7,4	52,35
21	Paso de estructuras por referencia	26,05	5,05	80,61	22,79	2,35	89,69
22	Aliasing de punteros	9,45	7,66	18,94	8,84	7,41	16,18
23	Cadenas de punteros	7,49	6,28	16,15	3,11	3,03	2,57
24	Pre-incremento vs post-incremento	451,36	464,08	0,00	51,47	52,85	0,00
25	Busqueda lineal	268,15	236,32	11,87	70,39	55,01	21,85
26	Estructuras IF en bucles	203,09	227,95	0,00	22,81	22,98	0,00

técnicas está especialmente recomendado en códigos que requieran intensos procesos de depuración.

Con objeto de facilitar la interpretación de la información relacionada en las tablas, se ofrecen las Figuras 5.22 y 5.23. De forma análoga a las gráficas relativas a IoT, en éstas las técnicas propuestas son también representadas en los ejes horizontales, mientras que los ejes verticales muestran el porcentaje de disminución del tiempo de ejecución logrado al aplicarlas (en comparación con los códigos estándar). Los tres nodos de cómputo utilizados se representan según sus procesadores: Intel Xeon E7-4830v3 (barras amarillas), E5-2660v3 (barras rojas) y E5-2670 (barras azules). La Figura 5.22 muestra los resultados cuando el compilador no realiza ninguna optimización (nivel 0), mientras que las mejoras alcanzadas cuando se usa el nivel máximo de optimización (3) pueden observarse en la Figura 5.23.

Los porcentajes de reducción conseguidos son similares en los tres nodos. Sin embargo, es importante resaltar que las diferencias producidas entre barras amarillas, rojas y azules respecto a determinadas técnicas no implican que un procesador tenga tiempos de ejecución más reducidos que otro, sino que el porcentaje de mejora obtenido al aplicar la técnica es menor en ese caso. De hecho, los procesadores que en un determinado experimento presentan tiempos de ejecución más reducidos que el resto, tienen menos posibilidades de mejorar aún más estos resultados al aplicar las técnicas.

Respecto a la escritura directa de códigos eficientes sin utilizar la optimización del compilador (ver Figura 5.22), todas las técnicas consiguen mejoras, once de ellas con reducciones de tiempo superiores al 30 %, y cuatro superiores al 69 %, demostrando su eficiencia cuando la optimización del compilador se establece por defecto (es decir, con el nivel 0).

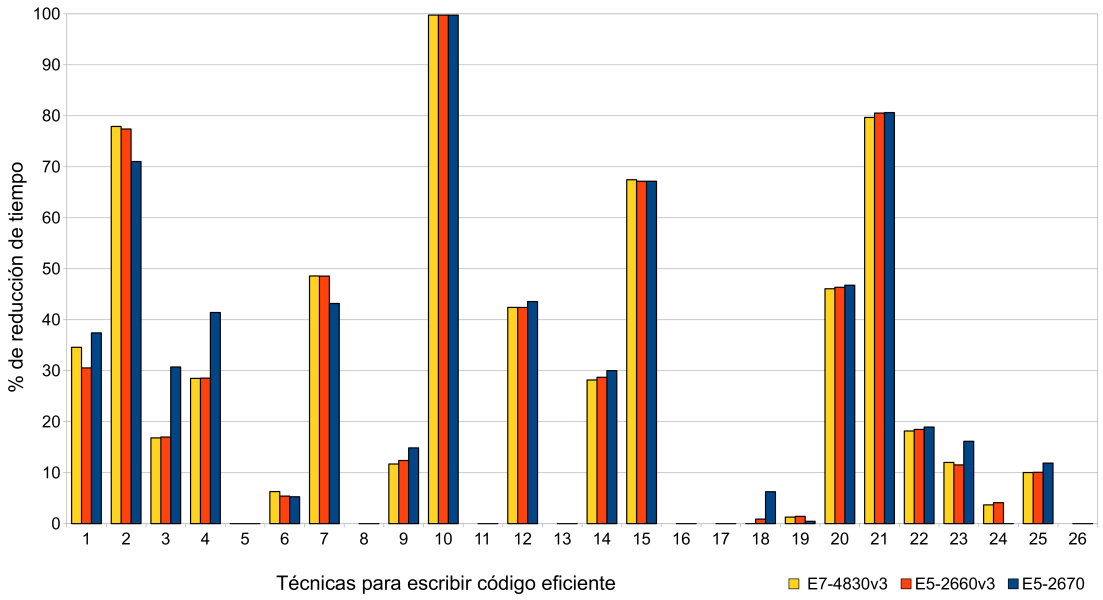

Figura 5.22: Porcentajes de reducción de tiempos de ejecución obtenidos mediante la escritura de código eficiente (sin optimización del compilador) en infraestructuras HPC.

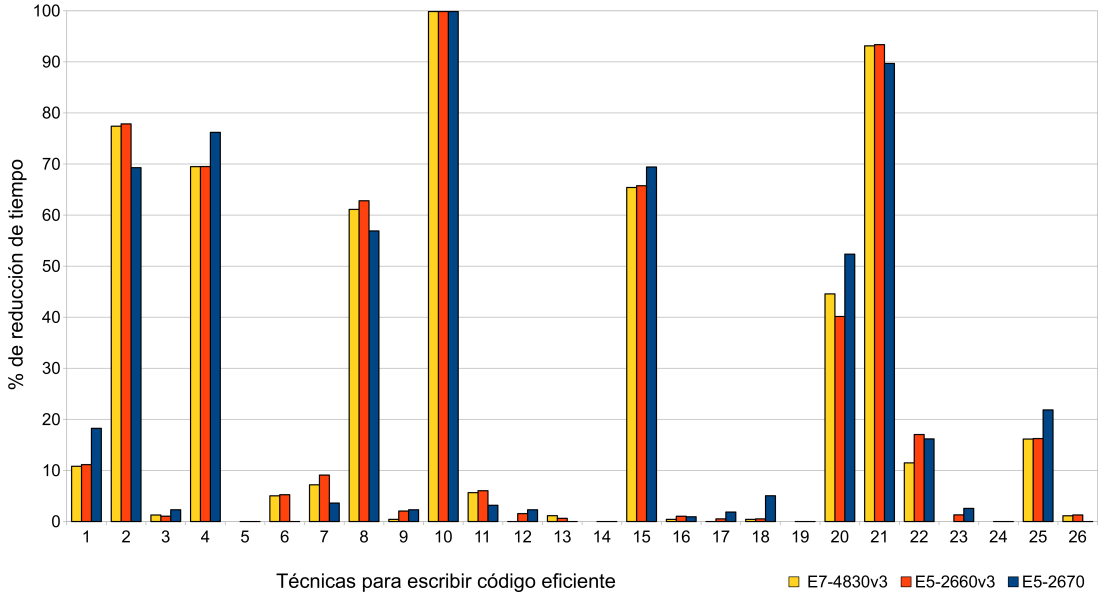

Figura 5.23: Porcentajes de reducción de tiempos de ejecución obtenidos mediante la escritura de código eficiente (con nivel 3 de optimización del compilador) en infraestructuras HPC.

También es importante señalar que el hecho de que una determinada técnica alcance un porcentaje de mejora menor con el nivel máximo de optimización que utilizando el nivel 0 (como en el caso de la técnica relativa a los *campos de bits* (T1)), no implica un aumento en el tiempo de ejecución cuando se elige el nivel 3, sino un menor impacto de la técnica al ser aplicada junto a la optimización automática proporcionada por el compilador (ver la primera barra amarilla de la Figura 5.22 y la primera barra amarilla de la Figura 5.23).

En cualquier caso, varias técnicas obtienen sus porcentajes de mejora más elevados cuando emplean el máximo nivel de optimización del compilador (ver Figura 5.23). Este incremento es particularmente destacable en las siguientes siete técnicas: *conjuntos de bits* (T2), *llamadas en cascada a funciones* (T4), *mapeo de estructuras* (T8), *control de excepciones* (T10), *inicialización frente a asignación* (T15), *desenrollado de bucles* (T20) y *paso de estructuras por referencia* (T21), con un índice de mejora medio del 73,04 %.

En resumen, gracias al uso de estas técnicas se pueden conseguir reducciones de hasta un 99,85 % en los tiempos de ejecución, demostrando así su eficacia para ser integradas en el transcompilador con objeto de detectar fragmentos de código cuyo rendimiento pueda ser mejorado por el usuario. Particularmente destacables son los porcentajes alcanzados con las técnicas de *control de excepciones* (T10) y *paso de estructuras por referencia* (T21), 99,85 % y 93,36 % respectivamente, como sucede en el caso de los dispositivos IoT y como era de esperar, ya que ambas son ampliamente conocidas. Las siguientes técnicas también obtuvieron notables mejoras: *conjuntos de bits* (T2), 77,85 %; *llamadas en cascada a funciones* (T4), 76,20 %; *mapeo de estructuras* (T8), 62,80 %; *inicialización frente a asignación* (T15), 69,42 %; y *desenrollado de bucles* (T20), 52,35 %. A continuación, se analizan con mayor profundidad estas siete técnicas, evaluando los factores que condicionan sus mejoras.

Conjuntos de bits (T2)

Los experimentos relacionados con esta técnica demuestran que en el ámbito del HPC también es recomendable utilizar conjuntos de *bits* en lugar de vectores cuando se requieren doce o más elementos booleanos (ver resumen del Código 5.4 -pág. 89- o Test C.2). La Figura 5.24 muestra la variación del porcentaje de mejora en función del número de elementos requeridos. Los experimentos desarrollados demuestran que no se obtiene ningún ahorro cuando esta técnica se aplica utilizando un tamaño inferior a 11. Sin embargo, a partir de 12 o más, el porcentaje crece de forma constante logrando mejoras de hasta el 70 % cuando se utilizan 30 o más elementos y de hasta el 90 % para 65 o más.

Para analizar esta técnica también se ha desarrollado una versión paralela del código estándar, distribuyendo el trabajo entre diferentes hilos de ejecución mediante una directiva OpenMP. Sin embargo, los tiempos obtenidos han sido significativamente mayores que utilizando la versión secuencial, debido al *overhead* incurrido cada vez que se invoca un bucle paralelo y se bifurcan los distintos hilos de ejecución. De este modo, el código eficiente que utiliza conjuntos de datos consigue mejoras de hasta un 100 % en comparación con la versión paralela.

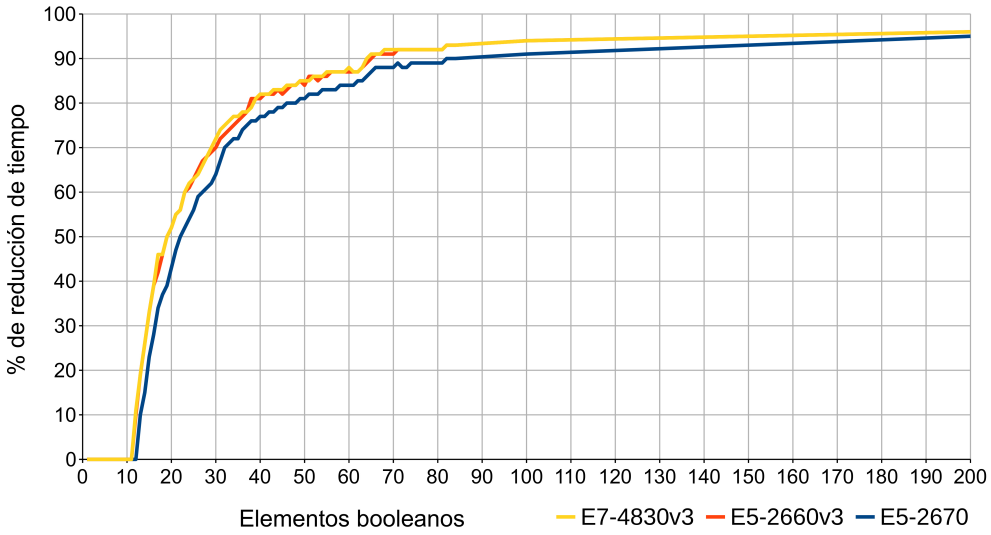

Figura 5.24: Conjuntos de bits (T2) en HPC. Porcentajes de mejora en los tiempos de ejecución según el número de elementos booleanos requeridos.

Llamadas en cascada a funciones (T4)

Como se indica en el apartado correspondiente relativo a dispositivos IoT, el porcentaje de mejora de esta técnica depende del tamaño del vector (ver Código 5.5 -pág. 90- o Test C.4), es decir, del número de iteraciones realizadas por el bucle. Especialmente destacable es el hecho de que con un vector de tan solo 3 elementos se consiguen reducciones del 60 % en los tiempos de ejecución (ver Figura 5.25). Con vectores de alrededor de 10 elementos se alcanzan mejoras superiores al 80 %, mientras que a partir de 200 o más éstas se estabilizan y permanecen constantes en torno al 95 %.

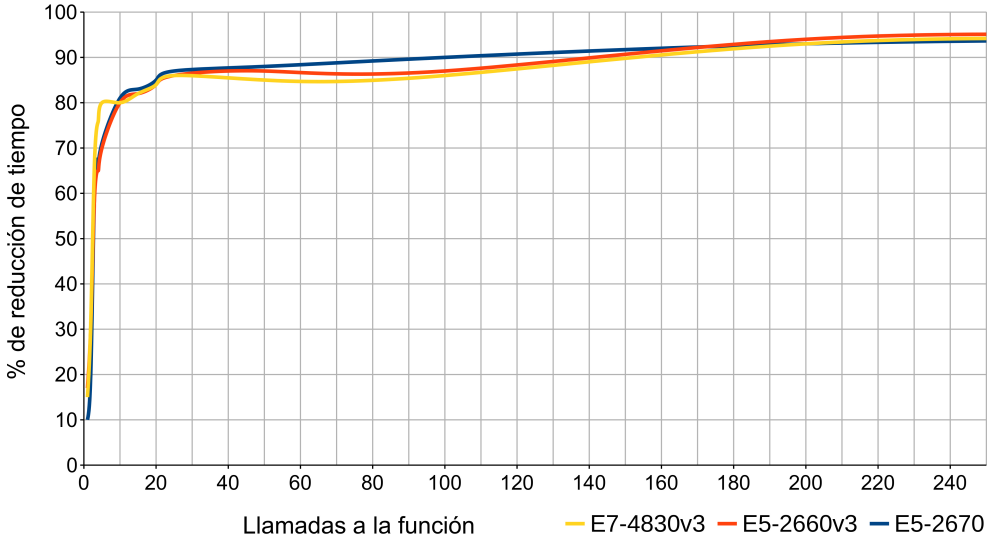

Figura 5.25: Llamadas en cascada a funciones (T4) en HPC. Porcentajes de mejora en los tiempos de ejecución según el número de llamadas a la función (número de iteraciones del bucle).

Al igual que en la técnica relativa a los *conjuntos de bits*, también se ha desarrollado una versión paralela del código estándar distribuyendo el trabajo entre diferentes hilos. Sin embargo, también en este caso, los tiempos de ejecución fueron especialmente peores que los obtenidos al utilizar la versión estándar secuencial, debido a las razones relacionadas con el *overhead* paralelo expuestas con anterioridad. Adicionalmente se ha implementado un tercer conjunto de experimentos consistentes en paralelizar el código eficiente, sin obtener resultados de consideración.

Mapeo de estructuras (T8)

Las mejoras obtenidas al aplicar esta técnica demuestran que, a diferencia de lo sucedido en los dispositivos IoT, es aconsejable utilizar tablas para mapear valores a constantes cuando se programan códigos destinados a infraestructuras HPC. Es decir, se recomienda recorrer las tablas mapeadas mediante bucles en lugar de utilizar declaraciones `if` anidadas para verificar cada uno de los casos (ver Código 5.13).

Los dos tests implementados transforman una cadena de caracteres en su valor numérico correspondiente, entre un total de 104 elementos posibles, de "1AAAAA" a "4ZZZZZ", y detienen sus búsquedas cuando encuentran el valor deseado. Para este propósito el código estándar emplea estructuras condicionales `if` anidadas, mientras que el código eficiente usa una tabla mapeada. Cabe destacar que, aunque existen

Código 5.13: Mapeo de estructuras (T8). Resumen comparativo de los tests.

```
1  static const struct {
2     const char data[7];
3     int        value;
4  }map[] = {
5     {"1AAAAA",1},
6     {"1BBBBB",2},
7       ...
8     {"4ZZZZZ",104}
9  };
10 int conversionToInt(const char *data) {
11    if      (strcmp(data,"1AAAAA")==0) return 1;
12    else if (strcmp(data,"1BBBBB")==0) return 2;
13    ...
14    else if (strcmp(data,"4ZZZZZ")==0) return 104;
15    else return -1; /* default case */
16 }
17 int conversionToInt2(const char *data) {
18    for (int i = 0; i < NELEMS(map); i++)
19       if (strcmp(data, map[i].data) == 0)
20           return map[i].value;
21    return -1; // default case
22 }
23 int standardTest() {
24   return conversionToInt("TTTTTT");
25 }
26 int efficientTest() {
27   return conversionToInt2("TTTTTT");
28 }
```

algoritmos más rápidos para trabajar con este tipo de mapeos, la búsqueda lineal es particularmente útil cuando se utilizan listas cortas. Además, se trata de un algoritmo muy conocido y especialmente recomendado para programadores noveles debido a su sencilla implementación.

Como se muestra en la Figura 5.26, el porcentaje de mejora al aplicar esta técnica depende del número de valores mapeados. Se consiguen reducciones de alrededor del 80 % cuando se utilizan menos de 10 elementos. Sin embargo, el rendimiento disminuye linealmente a medida que aumenta este número (considerando siempre el peor escenario, es decir, cuando el valor buscado se encuentra en la última posición de la tabla). La mejora es inferior al 50 % cuando existen más de 25 elementos y disminuye por debajo del 20 % a partir de 55 o más. Esta reducción en el rendimiento es debida principalmente a las dos comparaciones requeridas en cada iteración para encontrar un elemento en el mapa: la primera controla las iteraciones del bucle, mientras que la segunda verifica si el elemento actual coincide con el valor buscado. Asimismo, la aplicación de esta técnica siempre mejora los resultados de la utilización de estructuras `if` ampliamente anidadas, que en general deben ser evitadas.

Control de excepciones (T10)

Al igual que sucede con las reducciones de tiempo y consumo en los dispositivos IoT, esta técnica demuestra también su eficacia en los experimentos realizados sobre los nodos de cómputo (ver Código 5.7 -pág. 92- o Test C.10). Respecto a los resultados obtenidos destacan las mejoras del 99,85 % alcanzadas con solo 100 excepciones lanzadas (ver Figura 5.23 -pág. 115-). En consecuencia, también es aconsejable aplicar esta técnica en HPC para controlar errores inesperados dentro de bucles cuando existe una alta probabilidad de que estos ocurran. Como se mencionó anteriormente en relación a esta técnica, su alcance no está directamente relacionado tan solo con el número de excepciones lanzadas, sino también con la estructura de datos utilizada (de hecho, cabe recordar que la excepción implementada en los tests no presenta ningún contenido y únicamente consta de su

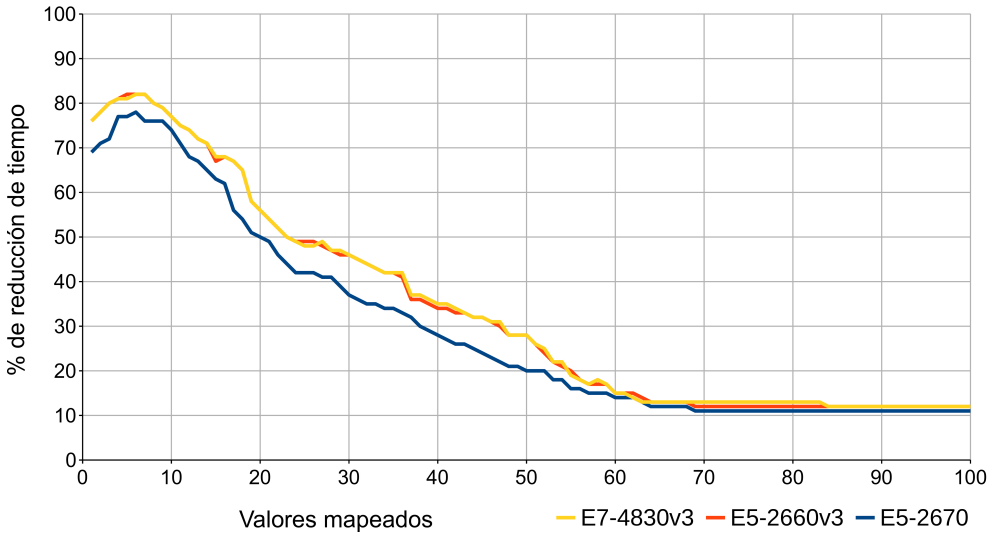

Figura 5.26: Mapeo de estructuras (T8) en HPC. Porcentajes de mejora en los tiempos de ejecución según el número de valores mapeados (buscando siempre el último elemento de la lista).

declaración). Así, su uso es recomendable independientemente del tipo de datos y, por tanto, existen infinitas variaciones posibles que pueden ser implementadas y evaluadas en este ámbito. Por este motivo se omite una evaluación en mayor profundidad, con el objetivo de ampliar próximamente el análisis de esta técnica.

Inicialización frente a asignación (T15)

La inicialización directa de las variables al ser declaradas también es recomendable en HPC, reduciendo notablemente los tiempos de ejecución al eliminar el efecto de las instancias y asignaciones posteriores (ver Código 5.8 -pág. 93 o Test C.15). En concreto, los experimentos demuestran que con esta técnica se alcanzan mejoras de hasta el 69,42 % (ver nuevamente Figura 5.23). Dado que la reducción obtenida depende del tipo de datos utilizados, en este caso también existen múltiples opciones posibles para implementar y evaluar su utilización, debiendo considerar tipos de datos primitivos, derivados y abstractos o definidos por el usuario, así como modificadores como *unsigned* o *long*. En consecuencia, se omite un análisis más detallado sobre las condiciones que influyen en las mejoras, que será abordado en próximos trabajos. En cualquier caso, la aplicación de esta técnica es aconsejable indistintamente del tipo de datos.

Desenrollado de bucles (T20)

A pesar del incremento en el tamaño del código que supone el desenrollado de un bucle, la consecuente disminución del número de iteraciones a realizar permite alcanzar mejoras del 52 % con matrices de 50 elementos. Además, dado que con los tests implementados los beneficios de aplicar esta técnica dependen del tamaño del vector y del número de instrucciones desenrolladas (ver Código 5.10 -pág. 95 o Test C.20), se han realizado varios experimentos adicionales con el objetivo de ofrecer un análisis pormenorizado.

La Figura 5.27 resume gráficamente los resultados de las pruebas con vectores de 50, 100, 200 y 300 elementos. En ella se representan los porcentajes de mejora alcanzados en base al número de iteraciones. De este modo, tomando como ejemplo un vector de 100 elementos, son necesarias 50 operaciones de asignación para inicializarlo completamente con dos únicas iteraciones del bucle, se requieren 25 asignaciones con 4 iteraciones y así sucesivamente hasta alcanzar las 50 iteraciones, que necesitan únicamente dos operaciones de asignación.

Los experimentos muestran que los porcentajes de mejora logrados son levemente superiores en el nodo de cómputo con el procesador ES-2670 (barras azules), donde los tiempos de ejecución también fueron ligeramente mayores y en consecuencia aumentan las posibilidades de mejorar los resultados aplicando la técnica propuesta.

En relación al vector de 100 elementos (Figura 5.27 a), se logran mejoras entre el 40 % y el 52 % cuando se realizan 10 iteraciones o menos y, por lo tanto, se desenrollan más de 5 operaciones de asignación. Asimismo, la mejora decrece cuando el número de iteraciones aumenta, situándose entre el 30 % y el 42 % con 25 iteraciones. Este hecho también se aplica al resto de vectores de mayor tamaño. Al superar las 25 iteraciones, la mejora tiende a desaparecer en los vectores de 100, 200 y 300 elementos (Figuras 5.27 b, c y d). Respecto al vector de 300 elementos (Figura 5.27 d), se obtiene una mejora de entre el

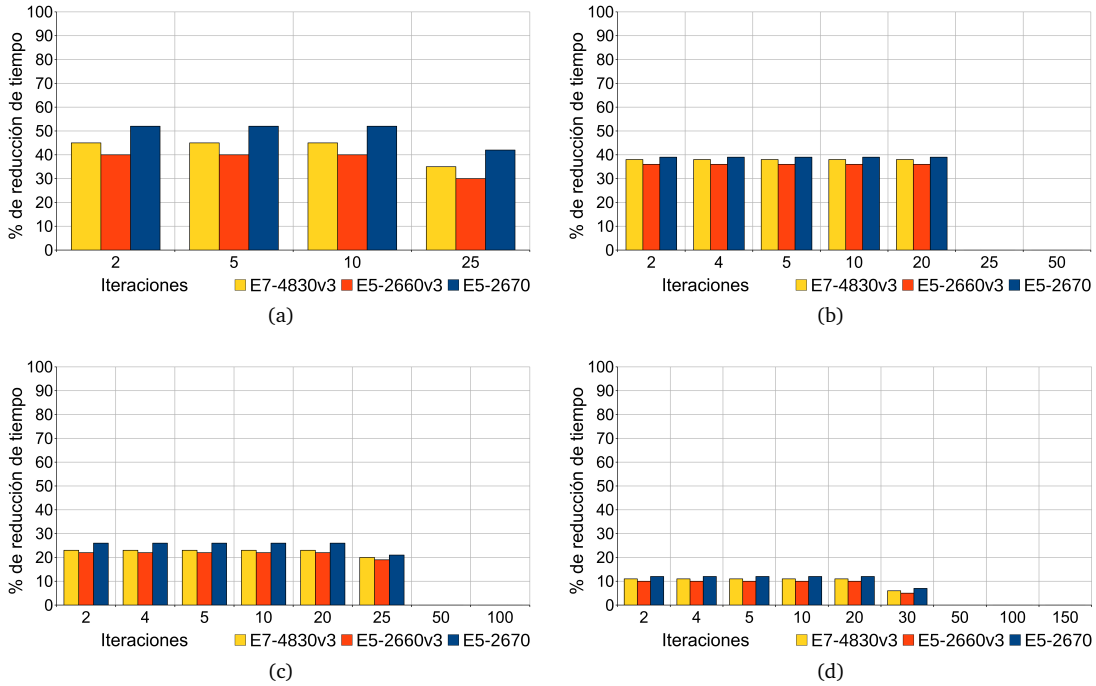

Figura 5.27: Desenrollado de bucles en HPC (T20). Porcentajes de mejora en los tiempos de ejecución según el número de iteraciones. Vectores de (**a**) 50 elementos, (**b**) 100 elementos, (**c**) 200 elementos y (**d**) 300 elementos.

10 % y el 12 % cuando el número de iteraciones es igual o inferior a 20. Por ello, la técnica debe aplicarse vigilando que el aumento en el número de operaciones desenrolladas no afecte al rendimiento de la caché. En consecuencia, es importante tener en cuenta que cuanto mayor es el vector recorrido, menor es la mejora alcanzada y además ésta tiende a desaparecer cuando el desenrollado involucra a más de 25 iteraciones.

De igual modo que con las técnicas de *conjuntos de bits* (T2) y *llamadas en cascada a funciones* (T4), también se ha desarrollado una versión paralela del código estándar para analizar la eficiencia de esta técnica. Sin embargo, al igual que en dichos experimentos, los tiempos de ejecución son significativamente peores que los alcanzados mediante la utilización de la versión estándar secuencial, nuevamente debido al *overhead* paralelo. Además, se ha ejecutado un tercer conjunto de pruebas que implementan la paralelización del código eficiente, con las cuales no se ha logrado mejorar los tiempos de ejecución.

Paso de estructuras por referencia (T21)

La aplicación de esta técnica, que implica el paso de estructuras por referencia para evitar el importante *overhead* originado al pasarlas por valor, ofrece mejoras de alrededor de un 89,69 %. Cabe recordar que en esta técnica los tests emplean una clase con dos atributos de tipo `string` y una subestructura formada por un vector de 10 elementos y un índice entero (ver Código 5.11 -pág. 96 o Test C.21).

De igual forma que en otras técnicas descritas anteriormente, existen numerosas opciones que pueden ser analizadas, dado que la mejora a alcanzar depende de las

estructuras de datos utilizadas. Por este motivo, se omite la realización de un análisis más detallado sobre las condiciones que influyen en estas ventajas, trabajo que será efectuado próximamente con objeto de integrar las técnicas en el transcompilador desarrollado. En cualquier caso, se recomienda igualmente aplicar esta propuesta, independientemente del tipo de datos.

De esta manera, se ha evaluado el impacto de las técnicas propuestas en esta tesis doctoral también en relación la computación de alto rendimiento, demostrando que compiladores como GCC son insuficientes para lograr el mejor rendimiento posible y no obtienen las mismas mejoras que un programador puede lograr si estas técnicas son aplicadas manualmente o mediante el transcompilador en el futuro. Además, tanto los experimentos realizados como los resultados alcanzados, son extrapolables a otras infraestructuras y procesadores de HPC. De hecho, las mejoras conseguidas son similares en los tres nodos de cómputo empleados en los experimentos, aunque en algunos casos existen algunas variaciones debido a las diferencias entre los procesadores. Asimismo, se ha demostrado que los usuarios de la computación paralela también deben ser conscientes del importante impacto que incluso pequeños fragmentos de código pueden tener en los tiempos de ejecución de sus programas. Consecuentemente, la aplicación de estas técnicas está especialmente recomendada para trabajos HPC cuyas ejecuciones requieren un destacable número de horas de CPU.

Capítulo 6

Conclusiones y trabajo futuro

En este capítulo se describen las conclusiones que se han alcanzado en relación a las propuestas realizadas en esta investigación, así como las futuras líneas de trabajo a abordar.

6.1. Conclusiones

La primera aportación de esta tesis doctoral ha consistido en ofrecer un transcompilador que paraleliza de forma automática códigos secuenciales. Su objetivo principal reside en que incluso usuarios noveles en computación paralela puedan hacer un uso más eficiente de los recursos disponibles en centros HPC, mejorando el rendimiento de sus ejecuciones y requiriendo menos tiempo para desarrollar sus algoritmos. Adicionalmente, el transcompilador propuesto facilita especialmente el aprendizaje inicial del modelo de programación, permitiendo a sus usuarios identificar de forma sencilla las partes secuenciales y paralelas de sus programas, así como las distintas directivas de paralelización generadas automáticamente. Para ello, el transcompilador examina, mediante análisis estático, programas escritos en lenguaje C/C++ y los transforma en programas OpenMP equivalentes, paralelizando sus bucles.

Con objeto de cubrir la necesidad de utilizar siempre una planificación de ejecución adecuada para cada código, se ha implementado un módulo de optimización que, mediante aprendizaje automático, predice la estrategia más apropiada en cada caso. Asimismo, en el marco de este optimizador se ha desarrollado también un sistema de apoyo a la toma de decisiones. Este DSS permite modificar, compilar y enviar los programas paralelizados a las colas de ejecución de las infraestructuras de cómputo. Además, posteriormente extrae de forma automática los resultados de rendimiento obtenidos en las ejecuciones, ordenando y analizando la información relativa a cada programa. Esto posibilita, a su vez, generar distintas versiones de cada código paralelo, con objeto de identificar la configuración con la que se obtienen los mejores resultados. De este modo, el DSS sirve también de apoyo a la creación de los conjuntos de entrenamiento requeridos por los algoritmos de aprendizaje automático utilizados.

Para demostrar la funcionalidad del transcompilador se han desarrollado diversos experimentos, basados en la paralelización de códigos secuenciales relacionados con problemas ampliamente conocidos y utilizados en diferentes ámbitos. Las pruebas han sido realizadas sobre dos procesadores distintos, ubicados en un nodo de cómputo de memoria compartida y en un clúster de memoria distribuida, que poseen unas características realmente representativas en sistemas de cómputo de alto rendimiento. Los resultados obtenidos han puesto de manifiesto las ventajas de utilizar el transcompilador, paralelizando de forma satisfactoria los programas correspondientes a cada uno de los experimentos y prediciendo correctamente la estrategia OpenMP más adecuada para cada código.

Como segundo objetivo principal de esta tesis, también se ha propuesto un conjunto de veintiséis técnicas de optimización destinadas a escribir código de programación eficiente para infraestructuras HPC. Su utilización persigue complementar las ventajas ofrecidas por el transcompilador. Estas técnicas han sido seleccionadas de la literatura con objeto de poder ser aplicadas en una gran variedad de rutinas y tareas de programación. Así, el propósito es reducir los tiempos de ejecución mediante su aplicación sobre los códigos a optimizar, conservando su semántica sin modificar los algoritmos.

Su eficiencia ha sido comprobada en un extenso conjunto de experimentos ejecutados en este caso en tres nodos de cómputo diferentes, con procesadores ampliamente utilizados en infraestructuras HPC de memoria compartida y distribuida. Cada experimento consta de dos *tests* desarrollados *ad hoc* para cada técnica, uno con código estándar y otro con el código eficiente resultante de aplicar la correspondiente técnica sobre el primero. Adicionalmente, se han analizado y evaluado con mayor profundidad las siete técnicas que han obtenido los resultados más notables, con la finalidad de analizar en detalle las condiciones que impactan en sus mejoras.

Las reducciones en los tiempos de ejecución logradas al aplicar las veintiséis técnicas han sido comparadas con las conseguidas mediante el uso de optimizaciones automáticas. Los resultados han demostrado que las características de compiladores como GCC se muestran insuficientes para aumentar el rendimiento del código. En concreto, no alcanzan las mismas mejoras que un programador puede lograr si aplica manualmente las técnicas propuestas en este trabajo. Además, varias de ellas han obtenido mejoras realmente destacables.

Los resultados analizados en esta investigación pueden ser extrapolados a otras infraestructuras y procesadores de HPC. De hecho, las mejoras conseguidas son similares en los tres nodos de cómputo utilizados para realizar los experimentos, aunque en algunos casos se produzcan ciertas variaciones debido a las diferencias existentes entre ellos.

Asimismo, se ha evidenciado que los usuarios que hacen uso de la computación de alto rendimiento, deben ser conscientes del importante impacto que incluso pequeños y simples fragmentos de código pueden tener en los tiempos de ejecución de sus desarrollos. Del mismo modo, la aplicación de estas técnicas está también especialmente recomendada en programas que emplean días o semanas de ejecución o requieren millones de horas de CPU al año.

Previamente a su aplicación sobre infraestructuras HPC, las técnicas han sido probadas en dispositivos del Internet de las cosas, con objeto de acotar adecuadamente los experimentos y reducir la complejidad de la investigación, especialmente respecto a las mediciones de ahorro energético. De esta forma, se ha evitado la utilización de un número elevado de horas de cómputo en los nodos HPC y se ha establecido el alcance de los distintos tests desarrollados. Las pruebas realizadas han confirmado que las técnicas también obtienen notables resultados con arquitecturas ARM y dispositivos IoT y en consecuencia, pueden ser extrapolables a un mayor número de sistemas computacionales. En concreto, su uso permite alcanzar importantes reducciones en los tiempos de ejecución y el consumo de energía de estos dispositivos. Así, se ha demostrado que el papel del programador también es fundamental a la hora de aumentar el ahorro energético. De igual forma que respecto a las infraestructuras HPC, también se han analizado con mayor detalle las siete técnicas que han alcanzado los mejores resultados en estos dispositivos, evaluando las condiciones que influyen en su rendimiento.

6.2. Limitaciones y trabajo futuro

Durante el desarrollo de las distintas propuestas y aportaciones realizadas, han surgido limitaciones y líneas de trabajo adicionales que serán abordadas en profundidad en futuras investigaciones, por ser consideradas fuera del alcance de esta tesis doctoral. Destacan principalmente las siguientes:

Relativas al transcompilador:

- Evaluación completa de las funcionalidades ofrecidas, mediante la utilización de un grupo significativo de usuarios de HPC, con diferentes habilidades y experiencias en distintas ramas científicas y tecnológicas.

- Aunque actualmente el transcompilador paraleliza programas escritos en lenguajes de programación C/C++, ha sido implementado de forma que pueda admitir otros lenguajes adicionales en próximos desarrollos.

- El transcompilador detecta dependencias entre instrucciones, determinando si es posible paralelizar cada bucle e informando al usuario en caso contrario. Sin embargo, no las corrige, debido a que sus capacidades se basan en el análisis estático del código. Por tanto, en estos casos, el usuario se ve obligado a reescribir manualmente el bucle o modificar el algoritmo de forma que se eviten estas dependencias. Además, en ocasiones no es posible aplicar este tipo de análisis, debido principalmente a patologías que no pueden ser resueltas en tiempo de compilación, incluyendo problemas de *aliasing* o accesos indirectos mediante punteros. Así, una de las principales líneas de desarrollo futuro consistirá en la implementación de análisis dinámicos en el transcompilador. No obstante, en primer lugar se prevé mejorar sus características actuales, de forma que pueda solucionar estas limitaciones obteniendo información adicional mediante la interacción del usuario.

- La funcionalidad y ventajas obtenidas en los resultados de las pruebas realizadas serían semejantes en otro tipo de recursos HPC. Sin embargo, se pretende ampliar la investigación considerando otras infraestructuras y arquitecturas computacionales, de forma que los resultados puedan ser extrapolados de forma más exacta a un mayor número de centros de supercomputación.
- Será necesario extender el conjunto de experimentos utilizados en el aprendizaje supervisado, tanto en el conjunto de entrenamiento, como para la propia validación.
- Deberán implementarse otros algoritmos de *machine learning* y comparar los resultados entre ellos.
- Se prevé ampliar el módulo de optimización considerando otras características correspondientes a los propios nodos de cómputo donde se realizan las ejecuciones, como por ejemplo, la carga del sistema en cada momento o las propiedades de su infraestructura, habida cuenta del significativo impacto que las particularidades de cada sistema presentan en los resultados.
- Se afrontará también la predicción automática del número de cores más adecuado para alcanzar un determinado *speedup* sin afectar significativamente a la eficiencia, con objeto de evitar desaprovechar los recursos de cómputo empleados.
- El método piloto de medición de *overheads* desarrollado será mejorado, para realizar un análisis más detallado de los efectos producidos, tanto por los incrementos en el número de cores de ejecución, como por la planificación paralela escogida en cada caso. Además, se analizará la utilización de modelos de rendimiento que permitan predecir con exactitud el tiempo empleado por las comunicaciones, ayudando así a mejorar el diseño y la implementación de los algoritmos.

Respecto a la aplicación de las técnicas:

- Ampliación de los experimentos desarrollados, para analizar el impacto de diversas modificaciones de las técnicas sobre el rendimiento de las mismas. De este modo, se pretende estudiar las variaciones en el tamaño y la dimensión de vectores y matrices, así como el uso de distintas estructuras de datos, considerando, entre otros, tipos de datos primitivos, derivados, abstractos o definidos por el usuario.
- Aplicación sobre otros dispositivos IoT, infraestructuras y arquitecturas de cómputo, con objeto de ampliar significativamente el alcance de la investigación.
- Implementación de un nuevo módulo en el transcompilador que permita integrar las técnicas de optimización propuestas, con objeto de detectar fragmentos de código cuyo rendimiento pueda ser mejorado gracias a la aplicación de alguna de estas técnicas.
- Análisis del ahorro energético en infraestructuras HPC al aplicar las técnicas propuestas. Como primera aproximación, se considerará el uso de simuladores que permitan reproducir el comportamiento en infraestructuras HPC y emplear *benchmarks* para analizar el consumo de energía.

Apéndices

Apéndice A

Publicaciones

A continuación se muestran las publicaciones realizadas por el autor en relación a esta tesis doctoral.

A.1. Revistas

1 Javier Corral-García, Felipe Lemus-Prieto and Miguel-Ángel Pérez-Toledano.
Efficient Code Development for Improving Execution Performance in High-Performance Computing Centers.
The Journal of Supercomputing (JCR Q2, IP 2.469), 77(4), pp 3261-3288. Springer, 30/07/2020.
ISSN: 0920-8542. DOI: https://doi.org/10.1007/s11227-020-03382-z.

2 Javier Corral-García, Felipe Lemus-Prieto, José-Luis González-Sánchez and Miguel-Ángel Pérez-Toledano.
Analysis of Energy Consumption and Optimization Techniques for Writing Energy- Efficient Code.
Electronics (JCR Q2, IP 2.412). 8(10) - 1192, pp. 1-21. MDPI, 19/10/2019.
ISSN: 2079-9292. DOI: https://doi.org/10.3390/electronics8101192.

3 Javier Corral-García, José-Luis González-Sánchez and Miguel-Ángel Pérez- Toledano.
Evaluation of Strategies for the Development of Efficient Code for Raspberry Pi Devices.
Sensors (JCR Q1, IP 3,031). 18(11) - 4066, pp. 1-21. MDPI, 21/11/2018.
ISSN: 1424-822. DOI: https://doi.org/10.3390/s18114066.

A.2. Congresos

1 Javier Corral-García, José-Luis González-Sánchez and Miguel-Ángel Pérez-Toledano.
Efficiency Analysis in Code Development for High-Performance Computing Centers.
Proceedings of the Seventh International Conference on Technological Ecosystems for Enhancing Multiculturality (TEEM 2019). ACM, pp. 539-547. León, España, 16-18/10/2019.
ISBN: 978-1-4503-7191-9.

2 Javier Corral-García, José-Luis González-Sánchez and Miguel-Ángel Pérez-Toledano.
Medición de overheads para el uso eficiente de recursos en centros de computación de alto rendimiento.
Avances en arquitectura y tecnología de computadores. Actas de las Jornadas SARTECO 2019. Universidad de Extremadura. pp. 326-333. Cáceres, España, 18-20/09/2019.
ISBN: 978-84-09-12127-4.

3 Javier Corral-García, José-Luis González-Sánchez and Miguel-Ángel Pérez-Toledano.
Compilador source-to-source, para la paralelización automática de códigos secuenciales, orientado a la gestión eficiente de recursos en centros de Computación de Alto Rendimiento.
Avances en arquitectura y tecnología de computadores. Actas de las Jornadas SARTECO 2018. Sociedad de Arquitectura y Tecnología de Computadores. pp. 321-328. Teruel, España, 12-14/09/2018.
ISBN: 978-84-09-04334-7.

4 Javier Corral-García, José-Luis González-Sánchez and Miguel-Ángel Pérez-Toledano.
Towards Automatic Parallelization of Sequential Programs and Efficient Use of Resources in HPC Centers.
Proceedings of the IEEE HPCS 2016 International Conference on High-Performance Computing & Simulation (Core B). pp. 947 - 954. Innsbruck, Austria, 18-22/07/2016.
ISBN: 978-1-5090-2087-4. DOI: 10.1109/HPCSim.2016.7568436.

5 Javier Corral-García, César Gómez-Martín, José-Luis González-Sánchez.
Green Code, Energy Efficiency in the Source Code for High-Performance Computing.
Actas de la 10ª Conferencia Ibérica de Sistemas y Tecnologías de la Información, CISTI 2015. II, pp. 61-64. Águeda, Aveiro, Portugal, 17-20/06/2015
ISBN: 978-989-98434-5-5.

6 César Gómez-Martín, Miguel-Ángel Vega-Rodríguez, José-Luis González-Sánchez, Javier Corral-García and David Cortés-Polo.
Performance and Energy Aware Scheduling Simulator for High-Performance Computing.
7th Iberian Grid Infrastructure Conference Proceedings, IBERGRID '13. pp. 17-29. Madrid, España. Editorial Universitat Politècnica de València, 19-20/09/2013.
ISBN: 978-84-9048-110-3

7 Javier Corral-García, José-Luis González-Sánchez, César Gómez-Martín and David Cortés-Polo.
Aplicabilidad de la reutilización de código a la implementación de soluciones en entornos HPC.
Atas da 8a Conferência Ibérica de Sistemas e Tecnologias de Informaçao, CISTI 2013. 2, pp. 359-362. Lisboa, Portugal, 19-22/06/2013.
ISBN: 978-989-98434-0-0.

8 Javier Corral-García, César Gómez-Martín, José-Luis González-Sánchez and David Cortés-Polo.
Development of Pattern-Based Scientific Applications for High-Performance Computing.
6th Iberian Grid Infrastructure Conference Proceedings. IBERGRID '12. pp. 147-158. Lisboa, Portugal, 7-9/11/2012.
ISBN: 978-989-98265-0-2.

9 Javier Corral-García, David Cortés-Polo, César Gómez-Martín and José-Luis González-Sánchez.
Methodology and Framework for the Development of Scientific Applications with High- Performance Computing through Web Services.
Proceedings of the 6th Euro American Conference on Telematics and Information Systems, EATIS 2012. pp. 173-180. ACM. Valencia, España, 23-25/05/2012.
ISBN: 978-1-4503-1012-3.

10 César Gómez-Martín, José-Luis González-Sánchez, Javier Corral-García, Ángel Bejarano-Borrega and Javier Lázaro-Jareño.
Performance Study of Hyper-Threading Technology on the LUSITANIA Supercomputer.
4th Iberian Grid Infrastructure Conference Proceedings, IBERGRID 2010. pp. 268-279. Braga, Portugal, 24-27/05/2010.
ISBN: 978-84-9745-549-7.

Apéndice B

Bucles paralelizados de forma automática en los experimentos

B.1. Conjunto de Mandelbrot

Código B.1: Bucle paralelo principal del conjunto de Mandelbrot.

```
#pragma omp parallel for num_threads(20) schedule(static) default(none)
  private(i, j, y1, x1, c, y, x, k, x2, y2)
  shared(m, n, count, b, y_min, y_max, g, x_min, x_max, r, count_max)

  for ( i = 0; i < m; i++ ) {
    for ( j = 0; j < n; j++ ) {
      x = ( ( double ) (     j - 1 ) * x_max
          + ( double ) ( m - j     ) * x_min )
          / ( double ) ( m     - 1 );

      y = ( ( double ) (     i - 1 ) * y_max
          + ( double ) ( n - i     ) * y_min )
          / ( double ) ( n     - 1 );

      count[i][j] = 0;

      x1 = x;
      y1 = y;

      for ( k = 1; k <= count_max; k++ ) {
        x2 = x1 * x1 - y1 * y1 + x;
        y2 = 2 * x1 * y1 + y;
        if ( x2 < -2.0 || 2.0 < x2 || y2 < -2.0 || 2.0 < y2 ) {
          count[i][j] = k;
          break;
        }
        x1 = x2;
        y1 = y2;
      }

      if ( ( count[i][j] %2 ) == 1 ) {
        r[i][j] = 255;
        g[i][j] = 255;
        b[i][j] = 255;
      }
      else {
        c = (int)(255.0*sqrt(sqrt(sqrt(
            ((double)(count[i][j])/(double)(count_max))))));
        r[i][j] = 3 * c / 5;
        g[i][j] = 3 * c / 5;
        b[i][j] = c;
      }
    }
  }
```

Referencias: [207, 213].

B.2. Cálculo de números primos

Código B.2: Bucle paralelo para el cálculo de números primos.

```
#pragma omp parallel for num_threads(20) schedule(static)
  private(i,end,prime)
  shared(n)
  reduction(+:j)
  reduction(+:total)

  for (i=1; i<=n; i+=2){
    end = (int) sqrt((float)i) + 1;
    prime = 1;
    j = 3;
    while (prime && (j <= end)){
       if (i %j == 0) prime = 0;
       j+= 2;
    }
    if (prime){
       //printf("%d\n",i);  // Se omite por cuestiones de rendimiento
       total++;
    }
}
```

B.3. Estimación de integral $\int_a^b f(x)\,\mathrm{d}x$

Código B.3: Bucle paralelo para el cálculo de la integral $\int_a^b f(x)\,\mathrm{d}x$.

```
#pragma omp parallel for num_threads(20) schedule(static)
  private(i,x)
  shared (a,b,n)
  reduction (+ : total)

  for (i=0.0; i<n; i++) {
    x = ((double)(n-i-1)*a + (double)(i)*b) / (double)(n-1);
    total = total + f(x);
  }

total = (b-a) * total / (double) n; // Estimación
error = fabs (total - exact);       // Error
```

B.4. Estimación de integral doble $\int_a^b f(x,y)\,\mathrm{d}x\mathrm{d}y$

Código B.4: Bucle paralelo para el cálculo de la integral doble $\int_a^b f(x,y)\,\mathrm{d}x\mathrm{d}y$.

```
1  #pragma omp parallel for num_threads(20) schedule(static) default(none)
2    private(i,x,y,j)
3    shared(b,a,nx,ny)
4    reduction(+: total)
5
6    for (i=1; i<=nx; i++) {
7      x = ((2*nx - 2*i + 1)*a + (2*i - 1)*b) / (2*nx);
8      for (j=1; j<=ny; j++) {
9        y = ((2*ny - 2*j + 1)*a + (2*j - 1)*b ) / (2*ny);
10       total = total + f ( x, y );
11     }
12   }
13
14 total = (b-a)*(b-a)*total / (double)(nx) / (double)(ny); // Estimación
15 error = fabs (total - exact);                            // Error
```

B.5. Cálculo de la Transformada Rápida de Fourier

Código B.5: Bucle paralelo para el cálculo de la Transformada Rápida de Fourier.

```
#pragma omp parallel for num_threads(20) schedule(static) default(none)
  private (ambr,ambu,j,ja,jb,jc,jd,jw,k,wjw)
  shared (a,b,c,d,lj,mj,mj2,sgn,w)

  for (j=0; j<lj; j++) {
    jw = j * mj;
    ja  = jw;
    jb  = ja;
    jc  = j * mj2;
    jd  = jc;

    wjw[0] = w[jw*2+0];
    wjw[1] = w[jw*2+1];

    if (sgn < 0.0) {
      wjw[1] = - wjw[1];
    }

    for (k=0; k<mj; k++) {
      c[(jc+k)*2+0] = a[(ja+k)*2+0] + b[(jb+k)*2+0];
      c[(jc+k)*2+1] = a[(ja+k)*2+1] + b[(jb+k)*2+1];

      ambr = a[(ja+k)*2+0] - b[(jb+k)*2+0];
      ambu = a[(ja+k)*2+1] - b[(jb+k)*2+1];

      d[(jd+k)*2+0] = wjw[0] * ambr - wjw[1] * ambu;
      d[(jd+k)*2+1] = wjw[1] * ambr + wjw[0] * ambu;
    }
  }
```

Referencias: [208, 209].

B.6. Factorización LU

Código B.6: Bucle paralelo para la factorización LU de matrices.

```
#pragma omp parallel for num_threads(20) schedule(static) default(none)
  private(i,j)
  shared(M,L,size,k)

  for (i=k+1; i<size; i++) {
    L[i][k] = M[i][k] / M[k][k];            // Paso de división
    for (j=k+1; j<size; j++)
      M[i][j] = M[i][j] - L[i][k]*M[k][j]; // Paso de eliminación
  }
```

B.7. Simulaciones de dinámica molecular

Código B.7: Bucle paralelo para simulaciones de dinámica molecular.

```
#pragma omp parallel for num_threads(20)
  private (i,j,k,rij,d,d2)
  shared (f,nd,np,pos,vel)
  reduction ( + : pe,ke)

  for (k=0; k<np; k++){
    // Calcula la energía y fuerzas potenciales
    for (i=0; i<nd; i++) {
      f[i+k*nd] = 0.0;
    }
    for (j=0; j<np; j++) {
      if (k!=j) {
        d = dist (nd,pos+k*nd,pos+j*nd,rij);
        // Atribuye la mitad de la energía potencial a la partícula J.
        if (d < PI2) {
          d2 = d;
        }
        else {
          d2 = PI2;
        }
        pe = pe + 0.5 * pow ( sin ( d2 ), 2 );
        for ( i = 0; i < nd; i++ ) {
          f[i+k*nd] = f[i+k*nd] - rij[i] * sin (2.0*d2) / d;
        }
      }
    }
    // Calcula la energía cinética
    for (i=0; i<nd; i++) {
      ke = ke + vel[i+k*nd] * vel[i+k*nd];
    }
  }
```

Referencia: [212].

B.8. Multiplicación de matrices densas

Código B.8: Bucle paralelo para la multiplicación de matrices densas.

```
#pragma omp parallel for num_threads(20) schedule(static) default(none)
  private(j,i,k)
  shared(n,a,l,b,c,m)

  for (j=0; j<n; j++) {
    for ( i = 0; i < l; i++ ) {
      a[i+j*l] = 0.0;
      for (k=0; k<m; k++) {
        a[i+j*l] = a[i+j*l] + b[i+k*l] * c[k+j*m];
      }
    }
  }
```

B.9. Aproximación de la ecuación de Poisson

Código B.9: Bucle paralelo para la aproximación de la ecuación de Poisson.

```
  #pragma omp parallel num_threads(20)
    shared  (dx, dy, f, itnew, itold, nx, ny, u, unew)
    private (i, it, j)

    // Método iterativo de Jacobi para resolver el sistema lineal

    for ( it = itold + 1; it <= itnew; it++ ) {

      #pragma omp for schedule(static) // Almacena la estimación actual
      for (j = 0; j < ny; j++ ) {
        for ( i = 0; i < nx; i++ ) {
          u[i][j] = unew[i][j];
        }
      }

      #pragma omp for schedule(static) // Realiza una nueva estimación
      for ( j = 0; j < ny; j++ ) {
        for ( i = 0; i < nx; i++ ) {
          if ( i == 0 || j == 0 || i == nx - 1 || j == ny - 1 ) {
            unew[i][j] = f[i][j];
          }
          else {
            unew[i][j] = 0.25 * (u[i-1][j] + u[i][j+1] + u[i][j-1]
                                 + u[i+1][j] + f[i][j] * dx * dy);
          }
        }
      }

    }
```

Apéndice C

Tests de técnicas de programación eficientes

Plantilla de test empleada en todos los experimentos

```
1
2  #include <cmath>
3  #include <iostream>
4  #include "watchtime.h"
5
6  #ifdef level
7     #define OPTIMIZE __attribute__((optimize(level)))
8  #else
9     #define OPTIMIZE
10 #endif
11
12 const int nms  = 10;        // number of measurements
13 const int reps = 100000000; // number of times a test is repeated
14                              //          (for a single measurement)
15
16 void stats(unsigned times[]){
17    unsigned totaltime,meantime,sumsq,sd;
18    float testtime;
19    totaltime = meantime = sumsq = 0;
20    printf("\t\t-------------------------------------------------\n");
21    for (int i=0; i<nms; i++){
22       printf("\t\tTime %d\t%u ms\n",i+1,times[i]);
23       totaltime += times[i];
24    }
25    meantime = totaltime / nms;
26    for (int i=0; i<nms; i++){
27       sumsq += (times[i] - meantime) * (times[i] - meantime);
28    }
29    sd = sqrt(sumsq / (nms-1));
30    testtime = ((float)meantime / (float)reps);
31    testtime *= 1000000; // ms to ns
32    printf("\t\t-------------------------------------------------\n");
33    printf("\t\tAverage measurement time: %u ms\n",meantime);
34    printf("\t\tStandard deviation: %u\n",sd);
35    printf("\t\t-------------------------------------------------\n");
36    printf("\t\tAverage test time: %.2f ns\n\n",testtime);
37 }
38
39
40 int main() {
41    int i,j;
42    unsigned times[nms],aux;
43    Watchtime time;
44
45    printf("\n\nTechnique 1: Bit fields\n\n");
46    printf("\tOptimization level %s\n\n",level);
47
48    printf("\tTest 1 (standard code)...\n\n");
49    for (i=0; i<nms; i++){
50       time.startTime();
51       for (j=0; j<reps; j++){
52          test1();
53       }
54       aux = time.getTime();
55       times[i] = aux;
56    }
```

```
57      stats(times);
58
59      printf("\tTest 2 (efficient code)...\n\n");
60      for (i=0; i<nms; i++){
61          time.startTime();
62          for (j=0; j<reps; j++){
63              test2();
64          }
65          aux = time.getTime();
66          times[i] = aux;
67      }
68      stats(times);
69  }
```

C.1. Campos de bits

```
1  typedef struct {
2    unsigned int bitA : 1;
3    unsigned int bitB : 1;
4    unsigned int bitC : 1;
5    unsigned int bitD : 1;
6  } BitField;
7
8  typedef struct {
9    unsigned int bits;
10 } IntegerBitField;
11
12 unsigned int OPTIMIZE getBitField(const BitField *d) {
13   return (d->bitA << 0) |
14          (d->bitB << 1) |
15          (d->bitC << 2) |
16          (d->bitD << 3);
17 }
18
19 unsigned int OPTIMIZE getIntegerBitField(const IntegerBitField *d) {
20   return d->bits;
21 }
22
23 unsigned int test1() {
24      BitField *p;
25      return getBitField(p);
26 }
27
28 unsigned int test2() {
29      IntegerBitField *p;
30      return getIntegerBitField(p);
31 }
```

C.2. Conjuntos de bits

```
1  void test1() {
2      bool array[32];
3      for (int i=0; i<32; i++){
4         array[i] = true;
5      }
6  }
7
8  void test2() {
9      std::bitset<32> bitset;
10     bitset.set();
11 }
```

C.3. Retorno de booleanos

```
1  bool OPTIMIZE getOR(bool argA,bool argB,bool argC,bool argD) {
2    return (argA || argB || argC || argD);
3  }
4
5  typedef unsigned int Flags;
6
7  #define flagA (1u << 0)
8  #define flagB (1u << 1)
9  #define flagC (1u << 2)
10 #define flagD (1u << 3)
11
12 bool OPTIMIZE getFlagsOR(Flags flags) {
13   return (flags & (flagA | flagB | flagC | flagD)) != 0;
14 }
15
16 void test1() {
17     getOR(1,1,0,0);
18 }
19
20 void test2() {
21     getFlagsOR(1100);
22 }
```

C.4. Llamadas en cascada a funciones

```
1  int const N = 20;
2  int a[N];
3
4  class Class {
5     private:
6        int data;
7     public:
8        void setData(int data);
9        int  getData();
10 };
11
12 void Class::setData(int data) {
13     Class::data = data;
14 }
15
16 int Class::getData(){
17     return data;
18 }
19
20 void OPTIMIZE test1(Class *c){
21     for (int i=0; i<N; i++){
22    if (c->getData()==1){
23            a[i] = 0;
24    }
25     }
26 }
27
28 void OPTIMIZE test2(Class *c){
29     int data = c->getData();
30     for (int i=0; i<N; i++){
31    if (data==1){
32            a[i] = 0;
33    }
34     }
35 }
```

C.5. Acceso por fila principal en matrices

```
1  int const N = 60;
2  int array[N][N];
3
4  void OPTIMIZE test1(){
5    for (int j=0; j<N; j++)
6        for (int i=0; i<N; i++)
7            array[i][j] = 0;
8  }
9
10 void OPTIMIZE test2(){
11   for (int i=0; i<N; i++)
12       for (int j=0; j<N; j++)
13           array[i][j] = 0;
14 }
```

C.6. Listas de inicialización

```
1  using namespace std;
2
3  class ClassA {
4     private:
5        string dataA;
6        string dataB;
7        int    dataC;
8        int    dataD;
9     public:
10       ClassA(string data1, string data2, int data3, int data4);
11 };
12
13 OPTIMIZE ClassA::ClassA(string data1, string data2, int data3, int data4) {
14     dataA = data1;
15     dataB = data2;
16     dataC = data3;
17     dataD = data4;
18 }
19
20 class ClassB {
21    private:
22       string dataA;
23       string dataB;
24       int    dataC;
25       int    dataD;
26    public:
27       ClassB(string data1, string data2, int data3, int data4);
28 };
29
30 OPTIMIZE ClassB::ClassB(string data1, string data2, int data3, int data4):
31         dataA(data1), dataB(data2), dataC(data3), dataD(data4) {
32 }
33
34 void OPTIMIZE test1(){
35     ClassA c("data1","data2",1,1);
36 }
37
38 void OPTIMIZE test2(){
39     ClassB c("data1","data2",1,1);
40 }
```

C.7. Eliminación de subexpresiones comunes

```
1  int x=100;
2  int i,j;
3
4  void OPTIMIZE test1(){
5      i = x + sqrt(16384) + 1;
6      j = x + sqrt(16384);
7  }
8
9  void OPTIMIZE test2(){
10     int aux = x + sqrt(16384);
11     i = aux + 1;
12     j = aux;
13 }
```

C.8. Mapeo de estructuras

```
#define NELEMS(a) ((int) (sizeof(a) / sizeof(a[0])))

static const struct {
    const char data[7]; /* NB. PIC */
    int        value;
} map[] = {
    { "AAAAAA", 1 },
    { "BBBBBB", 2 },
    { "CCCCCC", 3 },
    { "DDDDDD", 4 },
    { "EEEEEE", 5 },
    { "FFFFFF", 6 },
    { "GGGGGG", 7 },
    { "HHHHHH", 8 },
    { "IIIIII", 9 },
    { "JJJJJJ", 9 },
    { "KKKKKK", 9 },
    { "LLLLLL", 9 },
    { "MMMMMM", 9 },
    { "OOOOOO", 9 },
    { "PPPPPP", 9 },
    { "QQQQQQ", 9 },
    { "RRRRRR", 9 },
    { "SSSSSS", 9 },
    { "TTTTTT", 9 }
};

int OPTIMIZE dataToValue(const char *data) {
    if      (strcmp(data, "AAAAAA") == 0) return 1;
    else if (strcmp(data, "BBBBBB") == 0) return 2;
    else if (strcmp(data, "CCCCCC") == 0) return 3;
    else if (strcmp(data, "DDDDDD") == 0) return 4;
    else if (strcmp(data, "EEEEEE") == 0) return 5;
    else if (strcmp(data, "FFFFFF") == 0) return 6;
    else if (strcmp(data, "GGGGGG") == 0) return 7;
    else if (strcmp(data, "HHHHHH") == 0) return 8;
    else if (strcmp(data, "IIIIII") == 0) return 9;
    else if (strcmp(data, "JJJJJJ") == 0) return 10;
    else if (strcmp(data, "KKKKKK") == 0) return 11;
    else if (strcmp(data, "LLLLLL") == 0) return 12;
    else if (strcmp(data, "MMMMMM") == 0) return 13;
    else if (strcmp(data, "NNNNNN") == 0) return 14;
    else if (strcmp(data, "OOOOOO") == 0) return 15;
    else if (strcmp(data, "PPPPPP") == 0) return 16;
    else if (strcmp(data, "QQQQQQ") == 0) return 17;
    else if (strcmp(data, "RRRRRR") == 0) return 18;
    else if (strcmp(data, "SSSSSS") == 0) return 19;
    else if (strcmp(data, "TTTTTT") == 0) return 20;
    else return -1; /* default case */
}

int OPTIMIZE dataToValue2(const char *data) {
    for (int i = 0; i < NELEMS(map); i++)
        if (strcmp(data, map[i].data) == 0)
            return map[i].value;
    return -1; // default case
```

```
57  }
58
59  int test1() {
60      return dataToValue("TTTTTT");
61  }
62
63  int test2() {
64      return dataToValue2("TTTTTT");
65  }
```

C.9. Eliminación de código muerto

```
1  int global;
2
3  void OPTIMIZE test1(){
4    int i;
5    i = 1;          // dead store
6    global = 1;     // dead store
7    global = 2;
8    return;
9    global = 3;     // unreachable
10 }
11
12 void OPTIMIZE test2(){
13   global = 2;
14   return;
15 }
```

C.10. Control de excepciones

```
1  using namespace std;
2
3  class myexception: public exception {
4  } myex;
5
6  int OPTIMIZE test1(){
7     int num = 100;
8     for (int i=0; i<1; i++){
9        try{
10            if (num == 100) {
11       throw myex;
12            }
13       } catch (exception& e){
14       }
15    }
16    return 0;
17 }
18
19 int OPTIMIZE test2(){
20    int num = 100;
21    for (int i=0; i<1; i++){
22           if (num != 100) {
23       continue;
24            }
25    }
26    return 0;
27 }
```

C.11. Variables globales en bucles

```
1  int const N = 20;
2  int a[N];
3  int sum;
4
5  void initializeArray(int N){
6    for (int i=0; i<N; i++){
7       a[i] = i;
8    }
9  }
10
11 void OPTIMIZE test1(){
12   sum = 0;
13   for (int i=0; i<N; i++)
14     sum += a[i];
15 }
16
17 void OPTIMIZE test2(){
18   int t = 0;
19   for (int i=0; i<N; i++)
20     t += a[i];
21   sum = t;
22 }
```

C.12. Funciones inline o insertadas

```
1  int OPTIMIZE add (int x, int y) {
2     return x + y;
3  }
4
5  int OPTIMIZE sub(int x, int y) {
6    return add (x, -y);
7  }
8
9  int OPTIMIZE sub2(int x, int y) {
10   return x + -y;
11 }
12
13 void test1(){
14    sub(10,5);
15 }
16
17 void test2(){
18    sub2(10,5);
19 }
```

C.13. Variables globales

```
1  int value;
2
3  int f() {
4     return 512;
5  }
6
7  void OPTIMIZE test1() {
8    for (int i=0; i<50; i++) {
9       value += f();
10   }
11 }
12
13 void OPTIMIZE test2() {
14   int aux = value;
15   for (int i=0; i<50; i++) {
16      aux += f();
17   }
18   value = aux;
19 }
```

C.14. Constantes en bucles

```
1  #define IDENTIFIER_A 2
2  #define IDENTIFIER_B 1
3  #define IDENTIFIER_C 3
4
5  typedef struct {
6    unsigned int value;
7  } Structure;
8
9  void aux1(int i){
10    i++;
11 }
12 void aux2(int i){
13    i++;
14 }
15 void aux3(int i){
16    i++;
17 }
18
19
20 void OPTIMIZE test1(int N) {
21   int i;
22   Structure t,*pt;
23   pt = &t;
24
25   pt->value &= IDENTIFIER_C;
26
27   for (i=0; i<N; i++) {
28     if (pt->value & IDENTIFIER_A)
29       aux1(i);
30     else if (pt->value & IDENTIFIER_B)
31       aux2(i);
32     else
33       aux3(i);
34    }
35 }
36
37 void OPTIMIZE test2(int N) {
38   int i;
39   Structure t,*pt;
40   pt = &t;
41
42   pt->value &= IDENTIFIER_C;
43
44   if (pt->value & IDENTIFIER_A) {
45     for (i=0; i<N; i++)
46         aux1(i);
47   }else if (pt->value & IDENTIFIER_B) {
48     for (i=0; i<N; i++)
49         aux2(i);
50   }else {
51     for (i=0; i<N; i++)
52         aux3(i);
53   }
54 }
```

C.15. Inicialización frente a asignación

```
1  void OPTIMIZE test1() {
2      std::complex<double> mycomplex;
3      mycomplex = (3.14);
4  }
5
6  void OPTIMIZE test2() {
7      std::complex<double> mycomplex(3.14);
8  }
```

C.16. División con denominadores potencias de 2

```
1  int OPTIMIZE divide (unsigned int i) {
2    return i / 1024;
3  }
4
5  int OPTIMIZE divide2 (unsigned int i) {
6    return i >> 10;
7  }
8
9  void test1(){
10    divide(100000000);
11 }
12
13 void test2(){
14    divide2(100000000);
15 }
```

C.17. Multiplicación con factores potencias de 2

```
1  int OPTIMIZE multiply (int i) {
2    return i * 1024;
3  }
4
5  int OPTIMIZE multiply2 (unsigned int i) {
6    return i >> 10;
7  }
8
9  void test1(){
10    multiply(100000000);
11 }
12
13 void test2(){
14    multiply2(100000000);
15 }
```

C.18. Integer frente a character

```
1  char OPTIMIZE sum_char(char a, char b, char c, char d, char e) {
2      return a+b+c+d+e;
3  }
4
5  int OPTIMIZE sum_int(int a, int b, int c, int d, int e) {
6      return a+b+c+d+e;
7  }
8
9  void test1(){
10     sum_char(1,2,3,4,5);
11 }
12
13 void test2(){
14     sum_int(1,2,3,4,5);
15 }
```

C.19. Bucles con cuenta regresiva

```
1  int const N = 100;
2  int a[N];
3
4  void OPTIMIZE test1(){
5    for (int i=0; i<N; i++) {
6        a[i]=i;
7    }
8  }
9
10 void OPTIMIZE test2(){
11    int i = N+1;
12    while (--i) {
13       a[i] = i;
14    }
15 }
```

C.20. Desenrollado de bucles

```
1  const int N = 50;
2  int array[N];
3
4  void OPTIMIZE initialization1(){
5     int i;
6     for (i=0; i<N; i++){
7       array[i] = 0;
8     }
9  }
10
11 void OPTIMIZE initialization2(){
12    int i;
13    for (i=0; i<N; i+=5){
14      array[i]   = 0;
15      array[i+1] = 0;
16      array[i+2] = 0;
17      array[i+3] = 0;
18      array[i+4] = 0;
19    }
20 }
21
22 void test1(){
23     initialization1();
24 }
25
26 void test2(){
27     initialization2();
28 }
```

C.21. Paso de estructuras por referencia

```
1  using namespace std;
2
3  typedef struct {
4      int array[10];
5      int value;
6  }  Structure;
7
8  class Class {
9     private:
10       string    data_a ;
11       string    data_b;
12       Structure structure;
13    public:
14       Class(string data1, string data2, int i);
15       int getIndex();
16 };
17
18 OPTIMIZE Class::Class(string data1, string data2, int i) {
19     data_a = data1;
20     data_b = data2;
21     structure.value = i;
22 }
23
24 int OPTIMIZE Class::getIndex() {
25     return structure.value;
26 }
27
28 int OPTIMIZE test1(Class value){
29     return value.getIndex();
30 }
31
32 int OPTIMIZE test2(Class *reference){
33     return reference->getIndex();
34 }
```

C.22. Aliasing de punteros

```
1  int a,b,c,d,e;
2  int *pa = &a;
3  int *pb = &b;
4  int *pc = &c;
5  int *pd = &d;
6  int *pe = &e;
7
8  void OPTIMIZE pointersA(int *t1, int *t2, int *t3, int *t4, int *step) {
9      *t1 += *step;
10     *t2 += *step;
11     *t3 += *step;
12     *t4 += *step;
13 }
14
15 void OPTIMIZE pointersB(int *t1, int *t2, int *t3, int *t4, int *step) {
16     int s = *step;
17     *t1 += s;
18     *t2 += s;
19     *t3 += s;
20     *t4 += s;
21 }
22
23 void test1() {
24      pointersA(pa,pb,pc,pd,pe);
25 }
26
27 void test2() {
28      pointersB(pa,pb,pc,pd,pe);
29 }
```

C.23. Cadenas de punteros

```
1  typedef struct { int a,b,c,d,e; } Values;
2  typedef struct { Values *values; } Structure;
3
4  Structure structure,*pstructure;
5  Values values;
6
7  void OPTIMIZE test1() {
8    structure.values = &values;
9    pstructure = &structure;
10
11   pstructure->values->a = 0;
12   pstructure->values->b = 0;
13   pstructure->values->c = 0;
14   pstructure->values->d = 0;
15   pstructure->values->e = 0;
16 }
17
18 void OPTIMIZE test2() {
19   structure.values = &values;
20   pstructure = &structure;
21
22   Values *aux = pstructure->values;
23   aux->a = 0;
24   aux->b = 0;
25   aux->c = 0;
26   aux->d = 0;
27   aux->e = 0;
28 }
```

C.24. Pre-incremento frente a pos-incremento

```
1  int const N = 200;
2  int array[N+1];
3
4  void OPTIMIZE test1(){
5     for (int i=0; i<N;){
6     array[i] = i++;
7     }
8  }
9
10 void OPTIMIZE test2(){
11    for (int i=0; i<N;){
12         array[i] = ++i;
13    }
14 }
```

C.25. Búsqueda lineal

```
1  int const N = 100;
2  int list [N];
3  int *plist;
4
5  void OPTIMIZE inicialize(int *list, int N){
6    for (int i=0; i<N; i++){
7       list[i] = i;
8    }
9  }
10
11 int OPTIMIZE search1(int *list, int N, int want) {
12   int i;
13   for (i = 0; i < N; i++)
14     if (list[i] == want)
15       return i;
16   return -1;
17 }
18
19 int OPTIMIZE search2(int *list, int N, int want) {
20   int i;
21   list[N] = want;
22   i = 0;
23   while (list[i] != want)
24     i++;
25   if (i == N)
26     return -1;
27   return i;
28 }
29
30 int test1(){
31    plist = list;
32    return search1(plist,N,98);
33 }
34
35 int test2(){
36    plist = list;
37    return search2(plist,N,98);
38 }
```

C.26. Estructuras IF en bucles

```
1  int x=0;
2
3  int const N = 100;
4  int a[N];
5  int b[N];
6
7  void OPTIMIZE test1(){
8    for (int i=0; i<N; i++)
9      if (x==1)
10       a[i] = 0;
11     else
12       b[i] = 0;
13 }
14
15 void OPTIMIZE test2(){
16   if (x==1)
17     for (int i=0; i<N; i++)
18       a[i] = 0;
19   else
20     for (int i=0; i<N; i++)
21       b[i] = 0;
22 }
```

REFERENCIAS

If I have seen further it is by
standing on the shoulders of giants.

Sir Isaac Newton
Letter to Robert Hooke (February 15, 1676)

[1] G. Florimbi, H. Fabelo, E. Torti, S. Ortega, M. Marrero, G.M. Callico, G. Danese, and F. Leporati. Towards Real-Time Computing of Intraoperative Hyperspectral Imaging for Brain Cancer Detection Using Multi-GPU Platforms. *IEEE Access*, 8:8485–8501, 2020.
[Citado en pág. 1].

[2] T. Nemoto, N. Futakami, M. Yagi, A. Kumabe, A. Takeda, E. Kunieda, and N. Shigematsu. Efficacy Evaluation of 2D, 3D U-Net Semantic Segmentation and Atlas-Based Segmentation of Normal Lungs Excluding the Trachea and Main Bronchi. *Journal of Radiation Research*, 61(1):257–264, 2019.
[Citado en pág. 1].

[3] C. Álvarez, J. Ginés, M.A. Santamarta, F. Martín, Á.M. Guerrero, F.J. Rodríguez, and V. Matellán. Using HPC as a Competitive Advantage in an International Robotics Challenge. In *High Performance Computing*, pages 103–114, Cham, 2021. Springer International Publishing.
[Citado en pág. 1].

[4] M. Chi, A. Plaza, J.A. Benediktsson, Z. Sun, J. Shen, and Y. Zhu. Big Data for Remote Sensing: Challenges and Opportunities. *Proceedings of the IEEE*, 104(11):2207–2219, 2016.
[Citado en pág. 1].

[5] D.A.G. Oliveira, P. Rech, H.M. Quinn, T.D. Fairbanks, L. Monroe, S.E. Michalak, C. Anderson, P.O.A. Navaux, and L. Carro. Modern GPUs Radiation Sensitivity Evaluation and Mitigation Through Duplication With Comparison. *IEEE Transactions on Nuclear Science*, 61(6):3115–3122, 2014.
[Citado en pág. 1].

[6] A. Fernández, C. Fernández, J.A. Miguel, M.Á. Conde, and V. Matellán. Supercomputers in the Educational Process. In *Proceedings of the Seventh International Conference on Technological Ecosystems for Enhancing Multiculturality*, TEEM'19, pages 548–553, New York, NY, USA, 2019. Association for Computing Machinery.
[Citado en pág. 1].

[7] F.A. Lara, C. Neri, and H. Cota. Parallel Programming in Computing Undergraduate Courses: a Systematic Mapping of the Literature. *IEEE Latin America Transactions*, 17(08):1371–1381, 2019.
[Citado en pág. 1].

[8] M.U. Ashraf, F. Alburaei, A. Ahmad, and A. Algarni. Performance and Power Efficient Massive Parallel Computational Model for HPC Heterogeneous Exascale Systems. *IEEE Access*, 6:23095–23107, 2018.
[Citado en pág. 2].

[9] The OpenMP Application Program Interface Specification for Parallel Programming. http://openmp.org, 2021.
[Citado en pág. 2, 5 y 24].

[10] C. Yang, C. Huang, and C. Lin. Hybrid CUDA, OpenMP, and MPI Parallel Programming on Multicore GPU Clusters. *Computer Physics Communications*, 182(1):266–269, 2011. Computer Physics Communications Special Edition for Conference on Computational Physics Kaohsiung, Taiwan, Dec 15-19, 2009.
[Citado en pág. 2 y 19].

[11] S. Chandrasekaran and G. Juckeland. *OpenACC for Programmers: Concepts and Strategies*. Addison-Wesley Professional, 2017.
[Citado en pág. 2 y 19].

[12] G. Almasi. *PGAS (Partitioned Global Address Space) Languages*, pages 1539–1545. Springer US, Boston, MA, 2011.
[Citado en pág. 2, 19 y 20].

[13] GCC, the GNU Compiler Collection. https://gcc.gnu.org, 2021.
[Citado en pág. 2, 20, 78, 80 y 81].

[14] X. Tian, A. Bik, M. Girkar, P. Grey, H. Saito, and E. Su. Intel OpenMP C++/Fortran Compiler for Hyper-Threading Technology: Implementation and Performance. *Intel Technology Journal*, 6(1), 2002.
[Citado en pág. 2 y 22].

[15] CénitS: Centro Extremeño de iNvestigación, Innovación Tecnológica y Supercomputación. http://www.cenits.es, 2021.
[Citado en pág. 2 y 110].

[16] J.F. Bermejo-Martín, J. Calle-Cancho, J. Corral-García, D. Cortés-Polo, J.L. González-Sánchez, L.I. Jiménez Gil, and F. Lemus-Prieto. *Memoria Anual COMPUTAEX 2019*. Fundación COMPUTAEX, 2020.
[Citado en pág. 2].

[17] J.F. Bermejo-Martín, J. Calle-Cancho, J. Corral-García, D. Cortés-Polo, J.L. González-Sánchez, L.I. Jiménez Gil, and F. Lemus-Prieto. *Memoria conmemorativa del X Aniversario de CÃ©nitS-COMPUTAEX*. Fundación COMPUTAEX, 2020.
[Citado en pág. 2].

[18] Scayle (Supercomputación de Castilla y León). Proyectos de I+D+i. https://www.scayle.es/proyectos, 2021.
[Citado en pág. 2].

[19] J.M. Andión, M. Arenaz, G. Rodríguez, and J. Tourino. A Novel Compiler Support for Automatic Parallelization on Multicore Systems. *Parallel Computing*, 39(9):442–460, 2013.
[Citado en pág. 2, 3, 19, 20, 21 y 22].

[20] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. Association for Computing Machinery.
[Citado en pág. 2, 3 y 21].

[21] S. Baghdadi, A. Größlinger, and A. Cohen. Putting Automatic Polyhedral Compilation For GPGPU to Work. In *Proceedings of the 15th Workshop on Compilers for Parallel Computers (CPC'10)*, 2010.
[Citado en pág. 2, 3 y 21].

[22] M.M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-Cuda Code Generation for Affine Programs. In *International Conference on Compiler Construction*, pages 244–263. Springer, 2010.
[Citado en pág. 2, 3 y 21].

[23] M. Amini, B. Creusillet, S. Even, R. Keryell, O. Goubier, S. Guelton, J.O. McMahon, F.X. Pasquier, G. Péan, and P. Villalon. Par4all: From Convex Array Regions to Heterogeneous Computing. In *IMPACT 2012: Second International Workshop on Polyhedral Compilation Techniques HiPEAC 2012*, 2012.
[Citado en pág. 2, 3 y 21].

[24] N. Ventroux, T. Sassolas, A. Guerre, B. Creusillet, and R. Keryell. SESAM/Par4All: A Tool for Joint Exploration of MPSoC Architectures and Dynamic Dataflow Code Generation. In *Proceedings of the 2012 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, pages 9–16, 2012.
[Citado en pág. 2, 3 y 21].

[25] H. Bae, D. Mustafa, J-W.Lee, H. Lin, C. Dave, R. Eigenmann, S. Midkiff, et al. The Cetus Source-to-Source Compiler Infrastructure: Overview and Evaluation. *International Journal of Parallel Programming*, 41(6):753–767, 2013.
[Citado en pág. 2, 3 y 21].

[26] D. Quinlan, C. Liao, J. Too, R.P. Matzke, M. Schordan, and P. Lin. ROSE Compiler Infrastructure, 2012.
[Citado en pág. 2, 3 y 22].

[27] M. Palkowski and W. Bielecki. TRACO Parallelizing Compiler. In *Soft Computing in Computer and Information Science*, pages 409–421. Springer, 2015.
[Citado en pág. 2, 3 y 23].

[28] M.W. Hall, S.P. Amarasinghe, B.R. Murphy, S-W. Liao, and M.S. Lam. Interprocedural Parallelization Analysis In SUIF. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(4):662–731, 2005.
[Citado en pág. 2, 3 y 23].

[29] B. Pugh and C-W. Tseng. *Languages and Compilers for Parallel Computing: 15th Workshop, LCPC 2002, College Park, MD, USA, July 25-27, 2002, Revised Papers*, volume 2481. Springer, 2005.
[Citado en pág. 2, 3 y 23].

[30] T. Grosser, A. Groesslinger, and C. Lengauer. Polly Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. *Parallel Processing Letters*, 22(04):1250010, 2012.
[Citado en pág. 2, 3 y 23].

[31] S. Prema, R. Nasre, R. Jehadeesan, and B. Panigrahi. A Study on Popular Auto-Parallelization Frameworks. *Concurrency and Computation: Practice and Experience*, 31(17):e5168, 2019.
[Citado en pág. 2, 3, 21, 22 y 23].

[32] U. Bondhugula, S. Dash, O. Gunluk, and L. Renganarayanan. A Model for Fusion and Code Motion in an Automatic Parallelizing Compiler. In *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 343–352. IEEE, 2010.
[Citado en pág. 2 y 20].

[33] T. Sterling, M. Anderson, and M. Brodowicz. *High Performance Computing: Modern Systems and Practices*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2017.
[Citado en pág. 3, 7, 8, 9, 10 y 11].

[34] S. Dawson-Haggerty, A. Krioukov, and D.E. Culler. Power Optimization - A Reality Check. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-140*, 2009.
[Citado en pág. 3].

[35] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J.P. Fernandes, and J. Saraiva. Energy Efficiency Across Programming Languages: How Do Energy, Time, and Memory Relate? In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2017, pages 256–267, New York, NY, USA, 2017. ACM.
[Citado en pág. 3 y 34].

[36] S. Abdulsalam, D. Lakomski, Q. Gu, T. Jin, and Z. Zong. Program Energy Efficiency: The Impact of Language, Compiler and Implementation Choices. In *International Green Computing Conference*, pages 1–6, Nov 2014.
[Citado en pág. 3 y 31].

[37] C. Shore. Efficient C Code for ARM Devices. In *ARM Technology Conference*, pages 1–14, 2010.
[Citado en pág. 4].

[38] S. Garcia, D. Jeon, C.M. Louie, and M.B. Taylor. Kremlin: Rethinking and Rebooting Gprof for the Multicore Age. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 458–469, 2011. cited By 67.
[Citado en pág. 4 y 81].

[39] C. Curtsinger and E.D. Berger. Coz: Finding Code That Counts with Causal Profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 184–197, New York, NY, USA, 2015. Association for Computing Machinery.
[Citado en pág. 4 y 81].

[40] A. Yoga and S. Nagarakatte. A Fast Causal Profiler for Task Parallel Programs. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 15–26, New York, NY, USA, 2017. Association for Computing Machinery.
[Citado en pág. 4 y 81].

[41] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, and J. McDonald. *Parallel Programming in OpenMP*. Elsevier Science, 2000.
[Citado en pág. 5 y 25].

[42] S.F. Schifano, E. Luppi, D.A. Permadi, T. Nguyen, N. Nguyen, and L. Tomassetti. High Performance and Distributed Computing. *TORUS 1–Toward an Open Resource Using Services: Cloud Computing for Environmental Data*, pages 155–162, 2020.
[Citado en pág. 7].

[43] Benefits of the Supercomputing. Discover EuroHPC: The European High Performance Computing Joint Undertaking (EuroHPC JU). https://eurohpc-ju.europa.eu/discover-eurohpc, 2021.
[Citado en pág. 7].

[44] G. Kalbe. The European Approach to the Exascale Challenge. *Computing in Science Engineering*, 21(1):42–47, 2019.
[Citado en pág. 8].

[45] The European High Performance Computing Joint Undertaking (EuroHPC JU). https://eurohpc-ju.europa.eu/, 2021.
[Citado en pág. 8].

[46] HPCG, High Performance Conjugate Gradients. http://hpcg-benchmark.org, 2021.
[Citado en pág. 8].

[47] HPC AI500: A Benchmark Suite for HPC AI Systems. https://benchcouncil.org/HPCAI500, 2021.
[Citado en pág. 8].

[48] Z. Jiang, L. Wang, X. Xiong, W. Gao, C. Luo, F. Tang, C. Lan, H. Li, and J. Zhan. HPC AI500: The Methodology, Tools, Roofline Performance Models, and Metrics for Benchmarking HPC AI Systems. *CoRR*, abs/2007.00279, 2020.
[Citado en pág. 8].

[49] Graph 500 Benchmark. http://graph500.org/, 2021.
[Citado en pág. 8].

[50] Top 500 Supercomputer Sites. https://www.top500.org, 2020.
[Citado en pág. 8, 10, 12 y 110].

[51] J.J. Dongarra. Performance of Various Computers Using Standard Linear Equations Software. *ACM SIGARCH Computer Architecture News*, 18(1):17, 1990.
[Citado en pág. 8].

[52] J.J. Dongarra, P. Luszczek, and A. Petitet. The Linpack Benchmark: Past, Present and Future. *Concurrency and Computation: practice and experience*, 15(9):803–820, 2003.
[Citado en pág. 8].

[53] P.R. Luszczek, D.H. Bailey, J.J. Dongarra, J. Kepner, R.F. Lucas, R. Rabenseifner, and D. Takahashi. The HPC Challenge (HPCC) Benchmark Suite. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, volume 213, pages 1188455–1188677. Citeseer, 2006.
[Citado en pág. 8].

[54] A. Petitet, R.C. Whaley, J.J. Dongarra, and A. Cleary. HPL - A Portable Implementation of the High-Performance Linpack Benchmark For Distributed-Memory Computers.
http://www.netlib.org/benchmark/hpl, 2018.
[Citado en pág. 8].

[55] J.J. Dongarra. Report on the Fujitsu Fugaku System. *University of Tennessee-Knoxville Innovative Computing Laboratory, Tech. Rep. ICLUT-20-06*, 2020.
[Citado en pág. 8].

[56] J.J. Dongarra, P. Luszczek, and Y.M. Tsai. HPL - AI Mixed-Precision Benchmark.
https://icl.bitbucket.io/hpl-ai.
[Citado en pág. 8].

[57] T. Feng. Some Characteristics of Associative/Parallel Processing. In *Proc. of the 1972 Sagamore Computing Conference*, pages 5–16, 1972.
[Citado en pág. 9].

[58] W. Händler. The Impact of Classification Schemes on Computer Architecture. In *Advanced Computer Architecture*, pages 3–11. IEEE Computer Society Press, Washington, DC, USA, 1986.
[Citado en pág. 9].

[59] M.J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE transactions on computers*, 100(9):948–960, 1972.
[Citado en pág. 9].

[60] A. Halaas, B. Svingen, M. Nedland, P. Saetrom, O. Snove, and O.R. Birkeland. A recursive MISD architecture for pattern matching. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(7):727–734, 2004.
[Citado en pág. 9].

[61] P. Zardoshti, F. Khunjush, and H. Sarbazi-Azad. Adaptive Sparse Matrix Representation for Efficient Matrix–Vector Multiplication. *The Journal of Supercomputing*, 72(9):3366–3386, 2016.
[Citado en pág. 9].

[62] S. Kurgalin and S. Borzunov. *A Practical Approach to High-Performance Computing*. Springer, 2019.
[Citado en pág. 9, 12 y 33].

[63] G. Hager and G. Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, 2010.
[Citado en pág. 10, 24, 25, 28, 34 y 75].

[64] J.J. Dongarra, T. Sterling, H. Simon, and E. Strohmaier. High-Performance Computing: Clusters, Constellations, MPPs, and Future Directions. *Computing in Science & Engineering*, 7(2):51–59, 2005.
[Citado en pág. 11].

[65] C.E. Leiserson. Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing. *IEEE Transactions on Computers*, C-34(10):892–901, 1985.
[Citado en pág. 11].

[66] C. Chaintoutis, A. Bogris, and D. Syvridis. P-Torus: Wavelength-Based Switching in Packet Granularity for Intra-Data-Center Networks. *IEEE/OSA Journal of Optical Communications and Networking*, 11(9):491–500, 2019.
[Citado en pág. 11].

[67] F.J. Andújar, J.A. Villar, J.L. Sánchez, F.J. Alfaro, and J. Duato. N-dimensional twin torus topology. *IEEE Transactions on Computers*, 64(10):2847–2861, 2015.
[Citado en pág. 11].

[68] P. Faizian, M.A. Mollah, X. Yuan, Z. Alzaid, S. Pakin, and M. Lang. Random regular graph and generalized de bruijn graph with k -shortest path routing. *IEEE Transactions on Parallel and Distributed Systems*, 29(1):144–155, 2018.
[Citado en pág. 11].

[69] J. Kim, W.J. Dally, and D. Abts. Flattened Butterfly: A Cost-Efficient Topology for High-Radix Networks. In *Proceedings of the 34th annual international symposium on Computer architecture*, pages 126–137, 2007.
[Citado en pág. 11].

[70] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu. Dcell: A Scalable and Fault-Tolerant Network Structure for Data Centers. *SIGCOMM Comput. Commun. Rev.*, 38(4):75–86, August 2008.
[Citado en pág. 11].

[71] D. Lin, Y. Liu, M. Hamdi, and J. Muppala. Hyper-BCube: A Scalable Data Center Network. In *2012 IEEE International Conference on Communications (ICC)*, pages 2918–2923, 2012.
[Citado en pág. 11].

[72] F. Huang and J. Dai. Fast Data Dissemination in Kautz-Based Modular Datacenter Network. In *2012 International Conference on Systems and Informatics (ICSAI2012)*, pages 1606–1610, 2012.
[Citado en pág. 11].

[73] Y. Peng, Y. Yuan, X. Huang, W. Wu, and X. Meng. Research on Maintainability of Network Topology for Data Centers. In *2014 Sixth International Conference on Ubiquitous and Future Networks (ICUFN)*, pages 317–321, 2014.
[Citado en pág. 11].

[74] M. Besta, S.M. Hassan, S. Yalamanchili, R. Ausavarungnirun, O. Mutlu, and T. Hoefler. Slim Noc: A Low-Diameter on-Chip Network Topology for High Energy Efficiency and Scalability. *ACM SIGPLAN Notices*, 53(2):43–55, 2018.
[Citado en pág. 11].

[75] J. Kim, W.J. Dally, S. Scott, and D. Abts. Technology-Driven, Highly-Scalable Dragonfly Topology. In *2008 International Symposium on Computer Architecture*, pages 77–88. IEEE, 2008.
[Citado en pág. 11].

[76] A. Singla, C. Hong, L. Popa, and P.B. Godfrey. Jellyfish: Networking Data Centers Randomly. In *9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, pages 225–238, 2012.
[Citado en pág. 11].

[77] M. Besta, J. Domke, M. Schneider, M. Konieczny, S.D. Girolamo, T. Schneider, A. Singla, and T. Hoefler. High-Performance Routing With Multipathing and Path Diversity in Ethernet and HPC Networks. *IEEE Transactions on Parallel and Distributed Systems*, 32(4):943–959, 2021.
[Citado en pág. 11].

[78] G.F. Pfister. An Introduction to the Infiniband Architecture. *High performance mass storage and parallel I/O*, 42(617-632):102, 2001.
[Citado en pág. 11].

[79] C.B. Stunkel, R.L. Graham, G. Shainer, M. Kagan, S.S. Sharkawi, B. Rosenburg, and G.A. Chochia. The High-Speed Networks of the Summit and Sierra Supercomputers. *IBM Journal of Research and Development*, 64(3/4):3:1–3:10, 2020.
[Citado en pág. 11].

[80] M.S. Birrittella, M. Debbage, R. Huggahalli, J. Kunz, T. Lovett, T. Rimmer, K.D. Underwood, and R.C. Zak. Intel Omni-Path Architecture: Enabling Scalable, High Performance Fabrics. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pages 1–9. IEEE, 2015.
[Citado en pág. 11].

[81] W. Feng, P. Balaji, C. Baron, L.N. Bhuyan, and D.K. Panda. Performance Characterization of a 10-Gigabit Ethernet TOE. In *13th Symposium on High Performance Interconnects (HOTI'05)*, pages 58–63, 2005.
[Citado en pág. 11].

[82] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, and W. Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE micro*, 15(1):29–36, 1995.
[Citado en pág. 11].

[83] A.B. Yoo, M.A. Jette, and M. Grondona. SLURM: Simple Linux Utility for Resource Management. In *Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer Berlin Heidelberg, 2003.
[Citado en pág. 12].

[84] IBM Spectrum LSF Suites. https://www.ibm.com/products/hpc-workload-management/details, 2021.
[Citado en pág. 12].

[85] B. Nitzberg, J.M. Schopf, and J.P. Jones. *PBS Pro: Grid Computing and Scheduling Attributes*, pages 183–190. Kluwer Academic Publishers, USA, 2004.
[Citado en pág. 12].

[86] B.M. Bode, D.M. Halstead, R. Kendall, Z. Lei, and D. Jackson. The Portable Batch Scheduler and the Maui Scheduler on Linux Clusters. In *Annual Linux Showcase & Conference*, 2000.
[Citado en pág. 12].

[87] MOAB HPC suite. https://adaptivecomputing.com/cherry-services/moab-hpc-suite/, 2021.
[Citado en pág. 12].

[88] TORQUE Resource Manager. https://support.adaptivecomputing.com/torque-resource-manager-documentation, 2021.
[Citado en pág. 12].

[89] W. Gentzsch. Sun Grid Engine: Towards Creating a Compute Power Grid. In *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 35–36, 2001.
[Citado en pág. 12].

[90] I.T. Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley, 1995.
[Citado en pág. 13].

[91] G. Barlas. Multicore and Parallel Program Design. In Gerassimos Barlas, editor, *Multicore and GPU Programming*, pages 27–54. Morgan Kaufmann, Boston, 2015.
[Citado en pág. 14 y 16].

[92] T. Rauber and G. Rünger. *Parallel Programming for Multicore and Cluster Systems*. Springer-Verlag Berlin Heidelberg, 2013.
[Citado en pág. 15, 17 y 18].

[93] T. Rauber and G. Rünger. Exploiting Multiple Levels of Parallelism in Scientific Computing. In Michael K. Ng, Andrei Doncescu, Laurence T. Yang, and Tau Leng, editors, *High Performance Computational Science and Engineering*, pages 3–19, Boston, MA, 2005. Springer US.
[Citado en pág. 16].

[94] R. Rubinstein and D.P. Kroese. *Simulation and the Monte Carlo Method*, volume 10. John Wiley & Sons, 2016.
[Citado en pág. 17].

[95] S.K. Prasad, A. Gupta, A.L. Rosenberg, A.S., and C.C. Weems. *Topics in Parallel and Distributed Computing. Enhancing the Undergraduate Curriculum: Performance, Concurrency, and Programming on Modern Platforms*. Springer, 2018.
[Citado en pág. 17].

[96] G.M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *AFIPS Conference Proceedings - 1967 Spring Joint Computer Conference, AFIPS 1967*, pages 483–485, 1967.
[Citado en pág. 17].

[97] M.D. Hill and M.R. Marty. Amdahl's Law in the Multicore Era. *Computer*, 41(7):33–38, 2008.
[Citado en pág. 17].

[98] J.L. Gustafson. Reevaluating Amdahl's Law. *Communications of the ACM*, 31(5):532–533, 1988.
[Citado en pág. 18].

[99] Message Passing Interface Forum. https://www.mpi-forum.org/, 2021.
[Citado en pág. 19].

[100] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming With the Message-Passing Interface*, volume 1. MIT press, 1999.
[Citado en pág. 19].

[101] F. Bonelli, M. Tuttafesta, G. Colonna, L. Cutrone, and G. Pascazio. An MPI-Cuda Approach for Hypersonic Flows With Detailed State-to-State Air Kinetics Using a GPU Cluster. *Computer Physics Communications*, 219:178–195, 2017.
[Citado en pág. 19].

[102] N.P. Karunadasa and D.N. Ranasinghe. Accelerating High Performance Applications With CUDA and MPI. In *2009 International Conference on Industrial and Information Systems (ICIIS)*, pages 331–336, 2009.
[Citado en pág. 19].

[103] F. Bodin and S. Bihan. Heterogeneous Multicore Parallel Programming for Graphics Processing Units. *Scientific Programming*, 17(4):325–336, 2009.
[Citado en pág. 19].

[104] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.
[Citado en pág. 19].

[105] M. Zaharia, R.S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M.J. Franklin, et al. Apache Spark: A Unified Engine for Big Data Processing. *Communications of the ACM*, 59(11):56–65, 2016.
[Citado en pág. 19].

[106] H. Jin, D. Jespersen, P. Mehrotra, R. Biswas, L. Huang, and B. Chapman. High Performance Computing Using MPI and OpenMP on Multi-Core Parallel Systems. *Parallel Computing*, 37(9):562–575, 2011.
[Citado en pág. 19].

[107] R. Rabenseifner, G. Hager, and G. Jost. Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes. In *2009 17th Euromicro international conference on parallel, distributed and network-based processing*, pages 427–436. IEEE, 2009.
[Citado en pág. 19].

[108] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel Programmability and the Chapel Language. *The International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.
[Citado en pág. 19].

[109] R.W. Numrich and J. Reid. Co-Array Fortran for Parallel Programming. In *SIGPLAN Fortran Forum*, volume 17, pages 1–31, New York, NY, USA, August 1998. Association for Computing Machinery.
[Citado en pág. 19].

[110] M. Eleftheriou, S. Chatterjee, and J.E. Moreira. A C++ Implementation of the Co-Array Programming Model for Blue Gene/L. In *Parallel and Distributed Processing Symposium, International*, volume 3, pages 0105–0105. IEEE Computer Society, 2002.
[Citado en pág. 19].

[111] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. *Acm Sigplan Notices*, 40(10):519–538, 2005.
[Citado en pág. 19].

[112] Y. Zheng, A. Kamil, M.B. Driscoll, H. Shan, and K. Yelick. UPC++: A PGAS Extension for C++. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 1105–1114, 2014.
[Citado en pág. 19].

[113] R. Baghdadi, J. Ray, M.B. Romdhane, E. Del Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. Amarasinghe. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. In *Proceedings of the 2019 IEEEACM International Symposium on Code Generation and Optimization*, CGO 2019, pages 193–205, Piscataway, NJ, USA, 2019. IEEE Press.
[Citado en pág. 20].

[114] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.
[Citado en pág. 20].

[115] C. Lattner. LLVM and Clang: Next Generation Compiler Technology. In *The BSD conference*, volume 5, 2008.
[Citado en pág. 20].

[116] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. Pluto: A Practical and Fully Automatic Polyhedral Program Optimization System. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008)*. Citeseer, 2008.
[Citado en pág. 20].

[117] M.M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. A Compiler Framework for Optimization of Affine Loop Nests For GPGPUs. In *Proceedings of the 22nd annual international conference on Supercomputing*, pages 225–234, 2008.
[Citado en pág. 20].

[118] S. Verdoolaege, J.C. Juega, A. Cohen, J.I. GÓmez, C. Tenllado, and F. Catthoor. Polyhedral Parallel Code Generation For CUDA. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):1–23, 2013.
[Citado en pág. 20 y 21].

[119] M. Benabderrahmane, L.N. Pouchet, A. Cohen, and C. Bastoul. The Polyhedral Model Is More Widely Applicable Than You Think. In Rajiv Gupta, editor, *Compiler Construction*, pages 283–303, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
[Citado en pág. 20 y 21].

[120] K. Trifunovic, A. Cohen, D. Edelsohn, F. Li, T. Grosser, H. Jagasia, R. Ladelsky, S. Pop, J. Sjödin, and R. Upadrasta. Graphite Two Years After: First Lessons Learned From Real-World Polyhedral Compilation. In *GCC Research Opportunities Workshop (GROW'10)*, 2010.
[Citado en pág. 20].

[121] V. Basupalli, T. Yuki, S. Rajopadhye, A. Morvan, S. Derrien, P. Quinton, and D. Wonnacott. OMPVerify: Polyhedral Analysis for the OpenMP Programmer. In Barbara M. Chapman, William D. Gropp, Kalyan Kumaran, and Matthias S. Müller, editors, *OpenMP in the Petascale Era*, pages 37–53, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
[Citado en pág. 21].

[122] T. Yuki, P. Feautrier, S. Rajopadhye, and V. Saraswat. Array Dataflow Analysis for Polyhedral x10 Programs. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 23–34, 2013.
[Citado en pág. 21].

[123] P. Chatarasi, J. Shirako, M. Kong, and V. Sarkar. An Extended Polyhedral Model for SPMD Programs and Its Use in Static Data Race Detection. In Chen Ding, John Criswell, and Peng Wu, editors, *Languages and Compilers for Parallel Computing*, pages 106–120, Cham, 2017. Springer International Publishing.
[Citado en pág. 21].

[124] Polyhedral Compiler Collection (PoCC). http://web.cs.ucla.edu/ pouchet/software/pocc/, 2021.
[Citado en pág. 21].

[125] T. Yuki, G. Gupta, D. Kim, T. Pathan, and S. Rajopadhye. Alphaz: A System for Design Space Exploration in the Polyhedral Model. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 17–31. Springer, 2012.
[Citado en pág. 21].

[126] C. Chen, J. Chame, and M. Hall. CHiLL: A Framework for Composing High-Level Loop Transformations. Technical report, Citeseer, 2008.
[Citado en pág. 21].

[127] R. Baghdadi, J. Ray, M.B. Romdhane, E.D. Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. Amarasinghe. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 193–205, 2019.
[Citado en pág. 21 y 23].

[128] C. Bastoul. Efficient code generation for automatic parallelization and optimization. In *ISPDC*, volume 2, pages 23–30. Citeseer, 2003.
[Citado en pág. 21].

[129] C. Bastoul, A. Cohen, S. Girbal, S. Sharma, and O. Temam. Putting Polyhedral Loop Transformations to Work. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 209–225. Springer, 2003.
[Citado en pág. 21].

[130] H. Arabnejad, J. Bispo, J.G. Barbosa, and J.M.P. Cardoso. AutoPar-Clava: An Automatic Parallelization Source-to-Source Tool for C Code Applications. In *Proceedings of the 9th Workshop and 7th Workshop on Parallel Programming and RunTime Management Techniques for Manycore Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms*, PARMA-DITAM '18, pages 13–19, New York, NY, USA, 2018. Association for Computing Machinery.
[Citado en pág. 21, 22 y 24].

[131] U. Bondhugula, A. Acharya, and A. Cohen. The Pluto+ Algorithm: A Practical Approach for Parallelization and Locality Optimization of Affine Loop Nests. *ACM Trans. Program. Lang. Syst.*, 38(3), April 2016.
[Citado en pág. 21].

[132] A. Acharya and U. Bondhugula. PLUTO+: Near-Complete Modeling of Affine Transformations for Parallelism and Locality. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 54–64, New York, NY, USA, 2015. Association for Computing Machinery.
[Citado en pág. 21].

[133] B. Gaster, L. Howes, D.R. Kaeli, P. Mistry, and D. Schaa. *Heterogeneous Computing With OpenCL: Revised OpenCL 1*. Newnes, 2012.
[Citado en pág. 21].

[134] S. Lee, T. Johnson, and R. Eigenmann. Cetus–An Extensible Compiler Infrastructure for Source-to-Source Transformation. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 539–553. Springer, 2003.
[Citado en pág. 21].

[135] C. Dave, H. Bae, S. Min, S. Lee, R. Eigenmann, and S. Midkiff. Cetus: A Source-to-Source Compiler Infrastructure for Multicores. *Computer*, 42(12):36–42, 2009.
[Citado en pág. 21].

[136] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Taylor & Francis US, 2002.
[Citado en pág. 22].

[137] S. Prema, R. Jehadeesan, B. Panigrahi, and S. Murty. Dependency Analysis and Loop Transformation Characteristics of Auto-Parallelizers. In *Parallel Computing Technologies (PARCOMPTECH), 2015 National Conference on*, pages 1–6. IEEE, 2015.
[Citado en pág. 22].

[138] D.F. Bacon, S.L. Graham, and O.J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys (CSUR)*, 26(4):345–420, 1994.
[Citado en pág. 22].

[139] Intel Compilers. https://software.intel.com/compilers, 2021.
[Citado en pág. 22].

[140] D. Quinlan. ROSE: Compiler Support for Object-Oriented Frameworks. *Parallel processing letters*, 10(02n03):215–226, 2000.
[Citado en pág. 22].

[141] H. Arabnejad, J. Bispo, J.M.P. Cardoso, and J.G. Barbosa. Source-to-Source Compilation Targeting OpenMP-Based Automatic Parallelization of C Applications. *The Journal of Supercomputing*, 76, 09 2020.
[Citado en pág. 22].

[142] H. Arabnejad, J. Bispo, J.G. Barbosa, and J.M.P. Cardoso. An OpenMP based parallelization compiler for C applications. In *2018 IEEE Intl Conf on Parallel Distributed Processing with Applications, Ubiquitous Computing Communications, Big Data Cloud Computing, Social Computing Networking, Sustainable Computing Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom)*, pages 915–923, 2018.
[Citado en pág. 22].

[143] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. New User Interface for Petit and Other Extensions. *User Guide*, 1:996, 1996.
[Citado en pág. 22].

[144] S. Prema, R. Jehadeesan, and B. Panigrahi. A Comparative Study on Automatic Parallelisation Tools and Methods to Improve Their Usage. *International Journal of High Performance Computing and Networking*, 14(4):405–415, 2019.
[Citado en pág. 23].

[145] W. Pugh and E. Rosser. Iteration Space Slicing and Its Application to Communication Optimization. In *Proceedings of the 11th international conference on Supercomputing*, pages 221–228, 1997.
[Citado en pág. 23].

[146] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The Omega Library Interface Guide, 1995.
[Citado en pág. 23].

[147] V. Amaral, B. Norberto, M. Goulão, M. Aldinucci, S. Benkner, A. Bracciali, P. Carreira, E. Celms, L. Correia, C. Grelck, et al. Programming Languages for Data-Intensive HPC Applications: A Systematic Mapping Study. *Parallel Computing*, 91:102584, 2020.
[Citado en pág. 24 y 34].

[148] B. Chapman, G. Jost, and R. Van Der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*, volume 10. MIT press, 2008.
[Citado en pág. 25, 28, 39, 43 y 74].

[149] K. Kim, Y. Kim, and S. Park. A Probabilistic Machine Learning Approach to Scheduling Parallel Loops With Bayesian Optimization. *IEEE Transactions on Parallel and Distributed Systems*, 32(7):1815–1827, 2021.
[Citado en pág. 29, 30 y 76].

[150] S. Memeti, S. Pllana, A. Binotto, J. Kołodziej, and I. Brandic. Using Meta-Heuristics and Machine Learning for Software Optimization of Parallel Computing Systems: A Systematic Literature Review. *Computing*, 101(8):893–936, 2019.
[Citado en pág. 29].

[151] S. Memeti, S. Pllana, A. Binotto, J. Kołodziej, and I. Brandic. A Review of Machine Learning and Meta-Heuristic Methods for Scheduling Parallel Computing Systems. In *Proceedings of the International Conference on Learning and Optimization Algorithms: Theory and Applications*, LOPAL '18, New York, NY, USA, 2018. Association for Computing Machinery.
[Citado en pág. 29].

[152] Z. Wang and M. O'Boyle. Machine Learning in Compiler Optimization. *Proceedings of the IEEE*, 106(11):1879–1901, 2018.
[Citado en pág. 29].

[153] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M.F.P. O'Boyle, J. Thomson, M. Toussaint, and C.K.I. Williams. Using machine learning to focus iterative optimization. In *International Symposium on Code Generation and Optimization (CGO'06)*, pages 11 pp.–305, 2006.
[Citado en pág. 29 y 48].

[154] J. Cavazos and M.F.P. O'Boyle. Method-Specific Dynamic Compilation Using Logistic Regression. *SIGPLAN Not.*, 41(10):229–240, October 2006.
[Citado en pág. 29].

[155] S. Long, G. Fursin, and B. Franke. A Cost-Aware Parallel Workload Allocation Approach Based on Machine Learning Techniques. In *Proceedings of the 2007 IFIP International Conference on Network and Parallel Computing*, NPC'07, pages 506–515, Berlin, Heidelberg, 2007. Springer-Verlag.
[Citado en pág. 29 y 48].

[156] Z. Wang and M.F.P. O'Boyle. Mapping Parallelism to Multi-Cores: A Machine Learning Based Approach. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 75–84, New York, NY, USA, 2009. Association for Computing Machinery.
[Citado en pág. 29 y 30].

[157] D. Grewe and M.F.P. O'Boyle. A Static Task Partitioning Approach for Heterogeneous Systems Using OpenCL. In Jens Knoop, editor, *Compiler Construction*, pages 286–305, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
[Citado en pág. 29].

[158] G. Tournavitis, Z. Wang, B. Franke, and M.F.P. O'Boyle. Towards a Holistic Approach to Auto-Parallelization: Integrating Profile-Driven Parallelism Detection and Machine-Learning Based Mapping. *SIGPLAN Not.*, 44(6):177–187, June 2009.
[Citado en pág. 29, 30 y 48].

[159] X. Chen and S. Long. Adaptive Multi-versioning for OpenMP Parallelization via Machine Learning. In *2009 15th International Conference on Parallel and Distributed Systems*, pages 907–912, 2009.
[Citado en pág. 30 y 48].

[160] J.M. Keller, M.R. Gray, and J.A. Givens. A Fuzzy k-nearest Neighbor Algorithm. *IEEE transactions on systems, man, and cybernetics*, SMC-15:580–585, 1985.
[Citado en pág. 30 y 49].

[161] A. Qawasmeh, A.M. Malik, and B.M. Chapman. Adaptive OpenMP Task Scheduling Using Runtime APIs and Machine Learning. In *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*, pages 889–895, 2015.
[Citado en pág. 30 y 72].

[162] I. Rish. An Empirical Study of the Naive Bayes Classifier. In *IJCAI 2001 workshop on empirical methods in artificial intelligence*, volume 3, pages 41–46, 2001.
[Citado en pág. 30].

[163] M.W. Gardner and S. Dorling. Artificial Neural Networks (the Multilayer Perceptron). A Review of Applications in the Atmospheric Sciences. *Atmospheric environment*, 32(14-15):2627–2636, 1998.
[Citado en pág. 30].

[164] I. Steinwart and A. Christmann. *Support Vector Machines*. Springer Science & Business Media, 2008.
[Citado en pág. 30].

[165] M. Pal. Random Forest Classifier for Remote Sensing Classification. *International journal of remote sensing*, 26(1):217–222, 2005.
[Citado en pág. 30].

[166] Z. Khatami, L. Troska, H. Kaiser, J. Ramanujam, and A. Serio. HPX Smart Executors. In *Proceedings of the Third International Workshop on Extreme Scale Programming Models and Middleware*, ESPM2'17, New York, NY, USA, 2017. Association for Computing Machinery.
[Citado en pág. 30].

[167] C.P. Kruskal and A. Weiss. Allocating Independent Subtasks on Parallel Processors. *IEEE Transactions on Software Engineering*, SE-11(10):1001–1016, 1985.
[Citado en pág. 30].

[168] G. Laberge, S. Shirzad, P. Diehl, H. Kaiser, S. Prudhomme, and A.S. Lemoine. Scheduling Optimization of Parallel Linear Algebra Algorithms Using Supervised Learning. *2019 IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments (MLHPC)*, Nov 2019.
[Citado en pág. 30].

[169] J. Snoek, H. Larochelle, and R.P. Adams. Practical Bayesian Optimization of Machine Learning Algorithms. In *Proceedings of the 25th International Conference on Neural Information Processing Systems. Volume 2*, NIPS'12, pages 2951–2959, Red Hook, NY, USA, 2012. Curran Associates Inc.
[Citado en pág. 30].

[170] S.F. Hummel, E. Schonberg, and L.E. Flynn. Factoring: A Method for Scheduling Parallel Loops. *Commun. ACM*, 35(8):90–101, August 1992.
[Citado en pág. 30 y 76].

[171] A. Fog. Optimizing Software in C++–An Optimization Guide for Windows, Linux and Mac Platforms. Copenhagen University College of Engineering, 2018.
[Citado en pág. 31 y 54].

[172] A. Sloss, D. Symes, and C. Wright. *ARM System Developer's Guide: Designing and Optimizing System Software*. Elsevier, 2004.
[Citado en pág. 31, 54 y 57].

[173] L. Goldthwaite. Technical Report on C++ Performance. *ISO/IEC PDTR*, 18015, 2006.
[Citado en pág. 31 y 54].

[174] K. Guntheroth. *Optimized C++: Proven Techniques for Heightened Performance*. O'Reilly Media, Inc., 2016.
[Citado en pág. 31, 54, 56, 57 y 82].

[175] N. Malviya and A. Khunteta. Code Optimization Using Code Purifier. *International Journal of Computer Science and Information Technologies (IJCSIT), vol.6 (5)*, pages 4753–4757, 2015.
[Citado en pág. 31 y 54].

[176] N. Gupta, N. Seth, and P. Verma. Optimal Code Compiling in C. *International Journal of Computer Science and Information Technologies (IJCSIT), vol.6 (3)*, pages 2050–2057, 2015.
[Citado en pág. 31 y 54].

[177] K.D. Cooper, K.S. Mckinley, and L.Torczon. Compiler-Based Code-Improvement Techniques, 1998.
[Citado en pág. 31 y 54].

[178] M.E. Lee. Optimization of Computer Programs in C. http://leto.net/docs/C-optimization.php, 2018.
[Citado en pág. 32 y 54].

[179] K. Ghosh. Writing Efficient C and C Code Optimization.
https://www.codeproject.com/Articles/6154/Writing-Efficient-C-and-C-Code-Optimization, 2018.
[Citado en pág. 32 y 54].

[180] P. Isensee. C++ Optimization Strategies and Techniques.
http://www.tantalon.com/pete/cppopt/main.htm, 2018.
[Citado en pág. 32 y 54].

[181] D. Kim, J. Hong, I. Yoon, and S.Lee. Code Refactoring Techniques for Reducing Energy Consumption in Embedded Computing Environment. *Cluster Computing*, Nov 2016.
[Citado en pág. 32].

[182] J.J. Park, J. Hong, and S. Lee. Investigation for Software Power Consumption of Code Refactoring Techniques. In *SEKE*, pages 717–722, 2014.
[Citado en pág. 32].

[183] M. Gottschalk, J. Jelschen, and A. Winter. Energy-Efficient Code by Refactoring. *Softwaretechnik-Trends: Vol. 33, No. 2*, 2013.
[Citado en pág. 32].

[184] M. Geshi. *The Art of High Performance Computing for Computational Science, Vol. 1 Techniques of Speedup and Parallelization for General Purposes: Techniques of Speedup and Parallelization for General Purposes*. Springer, 01 2019.
[Citado en pág. 33 y 75].

[185] Clang: A C Language Family Frontend for LLVM. https://clang.llvm.org/, 2021.
[Citado en pág. 35].

[186] Java Compiler Compiler (JavaCC). The Java parser generator. https://javacc.org, 2021.
[Citado en pág. 35 y 36].

[187] S. Johnson et al. *Yacc: Yet Another Compiler-Compiler*, volume 32. Bell Laboratories Murray Hill, NJ, 1975.
[Citado en pág. 36].

[188] A.J. Dos Reis. *Compiler Construction Using Java, JavaCC, and Yacc*. John Wiley & Sons, 2012.
[Citado en pág. 36].

[189] M.E. Lesk and E. Schmidt. Lex: A Lexical Analyzer Generator, 1975.
[Citado en pág. 37].

[190] S. Gálvez and M.A. Mora. *Traductores y Compiladores con Lex/Yacc, JFlex/Cup y JavaCC*. Sergio Gálvez Rojas, 2005.
[Citado en pág. 37].

[191] P. Naur, J.W. Backus, F.L. Bauer, J. Green, C. Katz, and J. McCarthy. *Revised Report on the Algorithmic Language Algol 60*. State University of New York at Buffalo, Computing Center, 1976.
[Citado en pág. 37].

[192] K. Nakajima. Three-Level Hybrid vs. Flat MPI on the Earth Simulator: Parallel Iterative Solvers for Finite-Element Method. *Applied Numerical Mathematics*, 54(2):237–255, 2005. 6th IMACS.
[Citado en pág. 43].

[193] I.H. Witten, E. Frank, M.A. Hall, and C.J. Pal. The weka workbench online appendix. In *Data mining: practical machine learning tools and techniques*. Morgan Kaufmann Burlington, MA, 2016.
[Citado en pág. 47].

[194] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I.H. Witten. The WEKA Data Mining Software: An Update. *SIGKDD Explor. Newsl.*, 11(1):10–18, November 2009.
[Citado en pág. 47].

[195] T.A. Engel, A.S. Charão, M. Kirsch, and L. Steffenel. Performance Improvement of Data Mining in Weka Through Gpu Acceleration. *Procedia Computer Science*, 32:93–100, 2014.
[Citado en pág. 47].

[196] S. Wold, K. Esbensen, and P. Geladi. Principal Component Analysis. *Chemometrics and intelligent laboratory systems*, 2(1-3):37–52, 1987.
[Citado en pág. 48].

[197] R.O. Duda, P.E. Hart, et al. *Pattern Classification*. John Wiley & Sons, 2006.
[Citado en pág. 48].

[198] M. Stephenson and S. Amarasinghe. Predicting Unroll Factors Using Supervised Classification. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '05, pages 123–134, USA, 2005. IEEE Computer Society.
[Citado en pág. 48].

[199] I. Guyon and A. Elisseeff. An Introduction to Variable and Feature Selection. *Journal of machine learning research*, 3(Mar):1157–1182, 2003.
[Citado en pág. 48].

[200] ISO IEC. *ISO/IEC 14882:2020: Programming languages C++*. ISO (International Organization for Standardization), 2021.
[Citado en pág. 57].

[201] Javier Corral-García, José-Luis González-Sánchez, and Miguel-Ángel Pérez-Toledano. Towards automatic parallelization of sequential programs and efficient use of resources in HPC centers. In *2016 International Conference on High Performance Computing Simulation (HPCS)*, pages 947–954, july 2016.
[Citado en pág. 61].

[202] Javier Corral-García, José-Luis González-Sánchez, and Miguel-Ángel Pérez-Toledano. Compilador source-to-source, para la paralelización automática de códigos secuenciales, orientado a la gestión eficiente de recursos en centros de Computación de Alto Rendimiento. In *Avances en Arquitectura y Tecnología de Computadores, Jornadas SARTECO 2018*, pages 321–328, septiembre 2018.
[Citado en pág. 61 y 74].

[203] Intel Product Specifications. https://ark.intel.com/content/www/us/en/ark.html, 2021.
[Citado en pág. 62].

[204] Supercomputador LUSITANIA III.
http://www.cenits.es/lusitania-III, 2021.
[Citado en pág. 63].

[205] GNU Time. https://www.gnu.org/software/time, 2021.
[Citado en pág. 63].

[206] B.B. Mandelbrot. *The Fractal Geometry of Nature*, volume 1. WH freeman New York, 1982.
[Citado en pág. 63].

[207] J. Burkardt. Mandelbrot Image Using OpenMP. Department of Scientific Computing. Florida State University.
https://people.math.sc.edu/Burkardt/c_src/mandelbrot_openmp/mandelbrot_openmp.html, 2021.
[Citado en pág. 63 y 134].

[208] P. Arbenz and W. Petersen. *Introduction to Parallel Computing. A Practical Guide With Examples In C*. Oxford University Press, USA, 2004.
[Citado en pág. 65, 72 y 137].

[209] P. Arbenz and W. Petersen. Index of Software Examples for Book by W. Petersen and P. Arbenz. Introduction to Parallel Computing, a Practical Guide With Examples in C.
https://people.inf.ethz.ch/arbenz/book/, 2021.
[Citado en pág. 65 y 137].

[210] P. Michailidis and K.G. Margaritis. Implementing Parallel LU Factorization With Pipelining on a Multicore Using OpenMP. In *2010 13th IEEE International Conference on Computational Science and Engineering*, pages 253–260. IEEE, 2010.
[Citado en pág. 66].

[211] L. Verlet. Computer Experiments on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules. *Physical review*, 159(1):98, 1967.
[Citado en pág. 66].

[212] J. Burkardt. Molecular Dynamics. Department of Scientific Computing. Florida State University. https://people.math.sc.edu/Burkardt/c_src/md/md.html, 2021.
[Citado en pág. 66 y 138].

[213] J. Burkardt. Poisson Equation, Jacobi Iteration Parallelized With OpenMP. Department of Scientific Computing. Florida State University. https://people.sc.fsu.edu/~jburkardt/c_src/poisson_openmp/poisson_openmp.html, 2021.
[Citado en pág. 67 y 134].

[214] Y. Bazilevs and T. Hughes. Weak Imposition of Dirichlet Boundary Conditions in Fluid Mechanics. *Computers & Fluids*, 36(1):12–26, 2007.
[Citado en pág. 67].

[215] T.K. Ho. Random Decision Forests. In *Proceedings of 3rd international conference on document analysis and recognition*, volume 1, pages 278–282. IEEE, 1995.
[Citado en pág. 74].

[216] A. Vehtari, A. Gelman, and J. Gabry. Practical Bayesian Model Evaluation Using Leave-One-Out Cross-Validation and WAIC. *Statistics and computing*, 27(5):1413–1432, 2017.
[Citado en pág. 74].

[217] J.D. Rodriguez, A. Perez, and J.A. Lozano. Sensitivity Analysis of K-Fold Cross Validation in Prediction Error Estimation. *IEEE transactions on pattern analysis and machine intelligence*, 32(3):569–575, 2009.
[Citado en pág. 74].

[218] F.M. Ciorba, C. Iwainsky, and P. Buder. OpenMP Loop Scheduling Revisited: Making a Case for More Schedules. In Bronis R. de Supinski, Pedro Valero-Lara, Xavier Martorell, Sergi Mateo Bellido, and Jesus Labarta, editors, *Evolving OpenMP for Evolving Architectures*, pages 21–36, Cham, 2018. Springer International Publishing.
[Citado en pág. 76].

[219] Y. Zhang, M. Burcea, V. Cheng, R. Ho, and M. Voss. An Adaptive OpenMP Loop Scheduler for Hyperthreaded SMPs. In *ISCA PDCS*, pages 256–263. Citeseer, 2004.
[Citado en pág. 76].

[220] T.H. Tzen and L.M. Ni. Trapezoid Self-Scheduling: A Practical Scheduling Scheme for Parallel Compilers. *IEEE Transactions on parallel and distributed systems*, 4(1):87–98, 1993.
[Citado en pág. 76].

[221] S.F. Hummel, J. Schmidt, R.N. Uma, and J. Wein. Load-Sharing in Heterogeneous Systems via Weighted Factoring. In *Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, pages 318–328, 1996.
[Citado en pág. 76].

[222] Javier Corral-García, José-Luis González-Sánchez, and Miguel-Ángel Pérez-Toledano. Medición de overheads para el uso eficiente de recursos en centros de computación de alto rendimiento. In *Avances en Arquitectura y Tecnología de Computadores, Jornadas SARTECO 2019*, 8 pages, september 2019.
[Citado en pág. 76].

[223] J.A. Rico, J.C. Díaz, R.R. Manumachu, and A.L. Lastovetsky. A Survey of Communication Performance Models for High-Performance Computing. *ACM Comput. Surv.*, 51(6), January 2019.
[Citado en pág. 76].

[224] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash. Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications. *IEEE Communications Surveys Tutorials*, 17(4):2347–2376, Fourthquarter 2015.
[Citado en pág. 77].

[225] S. Hammoudi, Z. Aliouat, and S. Harous. Challenges and Research Directions for Internet of Things. *Telecommunication Systems*, 67(2):367–385, 2018.
[Citado en pág. 77].

[226] Raspberry Pi Foundation. https://www.raspberrypi.org, 2021.
[Citado en pág. 78, 79, 80 y 99].

[227] Arduino. https://www.arduino.cc/, 2021.
[Citado en pág. 78].

[228] N. Abbas, F. Yu, and Y. Fan. Intelligent Video Surveillance Platform for Wireless Multimedia Sensor Networks. *Applied Sciences*, 8(3), 2018.
[Citado en pág. 78].

[229] J.E. Noriega and J.M. Navarro Ruiz. On the Application of the Raspberry Pi as an Advanced Acoustic Sensor Network for Noise Monitoring. *Electronics*, 5(4), 2016.
[Citado en pág. 78].

[230] F. Leccese, M. Cagnetti, and D. Trinca. A Smart City Application: A Fully Controlled Street Lighting Isle Based on Raspberry-Pi Card, a ZigBee Sensor Network and WiMAX. *Sensors*, 14(12):24408–24424, 2014.
[Citado en pág. 78].

[231] J. García, L. Prieto, J. Pajares, M.M. Montalvo, and M.J.L. Boada. Real-Time Vehicle Roll Angle Estimation Based on Neural Networks in IoT Low-Cost Devices. *Sensors*, 18(7), 2018.
[Citado en pág. 78].

[232] J.L. Bayo, A. Martinez, W. Han, C. Fernandez, Y. Sun, and V. Traver. Wearable Sensors Integrated with Internet of Things for Advancing eHealth Care. *Sensors*, 18(6), 2018.
[Citado en pág. 78].

[233] W. Hajji and F.P. Tso. Understanding the Performance of Low Power Raspberry Pi Cloud for Big Data. *Electronics*, 5(2), 2016.
[Citado en pág. 78].

[234] B. Valle, T. Simonneau, R. Boulord, F. Sourd, T. Frisson, M. Ryckewaert, P. Hamard, N. Brichet, M. Dauzat, and A. Christophe. PYM: A New, Affordable, Image-Based Method Using a Raspberry Pi to Phenotype Plant Leaf Area in a Wide Diversity of Environments. *Plant methods*, 13(1):98, 2017.
[Citado en pág. 78].

[235] M. Kölling. Educational Programming on the Raspberry Pi. *Electronics*, 5(3), 2016.
[Citado en pág. 78].

[236] P. Abrahamsson, S. Helmer, N. Phaphoom, L. Nicolodi, N. Preda, L. Miori, M. Angriman, J. Rikkila, X. Wang, K. Hamily, et al. Affordable and Energy-Efficient Cloud Computing Clusters: The Bolzano Raspberry Pi Cloud Cluster Experiment. In *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on*, volume 2, pages 170–175. IEEE, 2013.
[Citado en pág. 78].

[237] Javier Corral-García, José-Luis González-Sánchez, and Miguel-Ángel Pérez-Toledano. Evaluation of Strategies for the Development of Efficient Code for Raspberry Pi Devices. *Sensors*, 18(11):4066, Nov 2018.
[Citado en pág. 79 y 84].

[238] Javier Corral-García, Felipe Lemus-Prieto, Miguel-Ángel Pérez-Toledano, and José-Luis González-Sánchez. Analysis of Energy Consumption and Optimization Techniques for Writing Energy-Efficient Code. *Electronics*, 8(10):1192, Oct 2019.
[Citado en pág. 79 y 100].

[239] Javier Corral-García, José-Luis González-Sánchez, and Miguel-Ángel Pérez-Toledano. Efficiency Analysis in Code Development for High-Performance Computing Centers. In *Proceedings of the Seventh International Conference on Technological Ecosystems for Enhancing Multiculturality*, TEEM'19, pages 539–547, New York, NY, USA, 2019. Association for Computing Machinery.
[Citado en pág. 79].

[240] Javier Corral-García, Felipe Lemus-Prieto, and Miguel-Ángel Pérez-Toledano. Efficient code development for improving execution performance in high-performance computing centers. *The Journal of Supercomputing*, Jul 2020.
[Citado en pág. 79 y 110].

[241] ARM Limited. *ARM Cortex-A72 MPCore Processor, Techinal Reference Manual*. ARM, 2016.
[Citado en pág. 94].

[242] Keith D. Cooper and Linda Torczon. Instruction Scheduling. In K.D. Cooper and L. Torczon, editors, *Engineering a Compiler (Second Edition)*, pages 639–677. Morgan Kaufmann, Boston, second edition edition, 2012.
[Citado en pág. 94].

[243] C. Gómez and M.A. Vega. Optimization of Resources in Parallel Systems Using a Multiobjective Artificial Bee Colony Algorithm. *The Journal of Supercomputing*, 74(8):4019–4036, 2018.
[Citado en pág. 110].

[244] C. Gómez, M.A. Vega, and J.L. González. Performance and Energy Aware Scheduling Simulator for HPC: Evaluating Different Resource Selection Methods. *Concurrency and Computation: Practice and Experience*, 27(17):5436–5459, 2015.
[Citado en pág. 110].

[245] NASA / Ames Research Center / NAS (United States). 32nd spot in TOP500 Supercomputer Sites, 54th edition. https://www.top500.org/site/48408, november 2019.
[Citado en pág. 110].

[246] Total Exploration Production (France). 41st spot in TOP500 Supercomputer Sites, 54th edition. https://www.top500.org/site/49546, november 2019.
[Citado en pág. 110].

[247] Government (United States). 62nd and 63rd spot in TOP500 Supercomputer Sites, 54th edition. https://www.top500.org/site/50046, november 2019.
[Citado en pág. 110].

[248] DOE/SC/Pacific Northwest National Laboratory (United States). 103rd spot in TOP500 Supercomputer Sites, 54th edition. https://www.top500.org/site/48611, november 2019.
[Citado en pág. 110].

[249] Air Force Research Laboratory (United States). 386th spot in TOP500 Supercomputer Sites, 54th edition. https://www.top500.org/site/49284, november 2019.
[Citado en pág. 110].

[250] Clemson University (United States). 392th spot in TOP500 Supercomputer Sites, 54th edition. https://www.top500.org/site/50100, november 2019.
[Citado en pág. 110].

[251] National Center for Atmospheric Research (NCAR) (United States). 439th spot in TOP500 Supercomputer Sites, 54th edition. https://www.top500.org/site/48418, november 2019.
[Citado en pág. 110].

[252] Service Provider T (China). 485th and 492nd spot in TOP500 Supercomputer Sites, 54th edition. https://www.top500.org/site/50329, november 2019.
[Citado en pág. 110].

Índice de abreviaturas

ANN *Artificial Neural Network*: red neuronal artificial. 30

API *Application Programming Interface*: interfaz de programación de aplicaciones. 19, 25, 26, 47

ARM *Advanced RISC (Reduced Instruction Set Computer) Machine*. 31, 77, 78, 80, 94, 109, 125

AST *Abstract Syntax Tree*: árboles de sintaxis abstracta. 20, 22, 35, 38, 39, 41, 42

CénitS Centro Extremeño de iNvestigación, Innovación Tecnológica y Supercomputación. 2, 62

ccNUMA *cache coherence NUMA*: acceso a memoria no uniforme con coherencia caché. 10

CFG *Control-Flow Graph Control Flow Graph*: grafo de control de flujo. 20, 22

CPU *Central Processing Unit*: unidad central de procesamiento. 2, 4, 31, 53, 80, 110, 112, 124

CSE *Common Subexpression Elimination*: eliminación de subexpresiones comunes. 31

CUDA *Compute Unified Device Architecture*: arquitectura unificada de dispositivos de cómputo. 19, 21

DDG *Data Dependence Graph*: gráfico de dependencia de datos. 20

DSM *Distributed Shared Memory*: memoria compartida distribuida. 10

DSS *Decision Support System*: sistema de apoyo a la toma de decisiones. 5, 35, 44–47, 49–51, 68, 70, 123

EBNF *Extended Backus-Naur Form*: notación extendida de Backus-Naur. 37

EuroHPC JU *European High-Performance Computing Joint Undertaking*: Empresa Común Europea de Informática de Alto Rendimiento. 8

FFT *Fast Fourier Transform*: transformada rápida de Fourier. 65

FLOPS *Floating Point Operations Per Second*: operaciones de coma flotante por segundo. 8

FSS *Factoring Self-Scheduling*. 30

GCC *GNU Compiler Collection*. 20, 21, 78, 80–82, 98, 111, 124

GPU *Graphics Processing Unit*: unidad de procesamiento gráfico. 9, 19, 21

HPC *High-Performance Computing*: computación de alto rendimiento. 1–5, 8, 11, 20, 31–33, 44, 47, 51, 62, 63, 77, 79, 110, 112, 116, 118–120, 123–126

HPL *High Performance Linpack*. 8

ICC *Intel C Compiler*. 22, 23

IoT *Internet of Things*: Internet de las cosas. 4, 61, 77–79, 84, 88, 98, 111, 112, 114, 116–119, 125, 126

JAR *Java ARchive*. 24

JavaCC *Java Compiler Compiler*. 35–38

JRE *Java Runtime Environment*. 24

k-nn *k-nearest neighbors*: k vecinos más próximos. 49

LSF *Load Sharing Facility*. 12

MIMD *Multiple Instruction, Multiple Data*: múltiples instrucciones, múltiples datos. 9

MISD *Multiple Instruction, Single Data*: múltiples instrucciones, un dato. 9

MPI *Message Passing Interface*: interfaz de paso de mensajes. 19, 25

MPP *Massively Parallel Processors*. 10

NUMA *Non-Uniform Memory Access*: acceso a memoria no uniforme. 10

OpenACC *for Open Accelerators*. 19

OpenMP *Open Multi-Processing*. 5, 14, 19, 21–26, 28–30, 35, 39–41, 43–47, 49, 50, 62, 67, 70, 74, 76, 116, 123, 124

PCA *Principal Component Analysis*: análisis de componentes principales. 48

PCAM *Partitioning, Communication, Agglomeration and Mapping*: particionado, comunicación, aglomeración y mapeo. 13

PDE *Partial Differential Equation*: ecuación en derivadas parciales. 67

PGAS *Partitioned Global Address Space*: espacio de direcciones global particionado. 19

PIPS *Parallelization Infrastructure for Parallel Systems*. 21

SCoPs *Static Control Part*. 20

SIMD *Single Instruction, Multiple Data*: una instrucción, múltiples datos. 9

SISD *Single Instruction, Single Data*: una instrucción, un dato. 9

Slurm *Simple Linux Utility for Resource Management*. 12, 45, 46, 112

SMP *Symmetric Multi-Processing*: multiprocesamiento simétrico. 10, 11

SVM *Support Vector Machine*: máquina de vectores de soporte. 30

UMA *Uniform Memory Access*: acceso a memoria uniforme. 10

WEKA *Waikato Environment for Knowledge Analysis*. 47

Este libro fue escrito entre los meses de enero y mayo de 2021, bajo la paciente e inseparable compañía de Kira, mi westie (West Highland White Terrier). A ella también va dedicada esta tesis.