Universidad de La Laguna

Escuela de Doctorado y Estudios de Posgrado

TÍTULO DE LA TESIS DOCTORAL

Extension del Modelo de Openmp a Memoria Distribuida

AUTOR/A

ANTONIO JAVIER	DORTA	LORENZO

DIRECTOR/A

FRANCISCO DE	SANDE	GONZALEZ

CODIRECTOR/A

DEPARTAMENTO O INSTITUTO UNIVERSITARIO

FECHA DE LECTURA

18/12/08

Departamento de Estadística,
Investigación Operativa y Computación

Extensión del modelo de OpenMP a memoria distribuida

D. Antonio Javier Dorta Lorenzo

2008

Extensión del modelo de OpenMP a memoria distribuida

Departamento de Estadística,
Investigación Operativa y Computación

Memoria de Tesis Doctoral para optar al grado de
Doctor en Informática por la Universidad de La Laguna

presentada por:

D. Antonio Javier Dorta Lorenzo

Director:

Dr. Francisco de Sande González

San Cristobal de La Laguna
septiembre de 2008

D. Francisco de Sande González, Profesor Titular de Universidad de Lenguajes y Sistemas Informáticos, adscrito al Departamento de Estadística, Investigación Operativa y Computación de la Universidad de La Laguna,

CERTIFICA:

Que la presente memoria titulada

Extensión del modelo de

OpenMP a memoria distribuida

ha sido realizada bajo mi dirección por el Ingeniero en Informática D. Antonio Javier Dorta Lorenzo, y constituye su Tesis para optar al grado de Doctor por la Universidad de La Laguna.

Y para que así conste, en cumplimiento de la legislación vigente, y a los efectos que haya lugar, firmo el presente en La Laguna, a cinco de septiembre de dos mil ocho.

Fdo: Dr. Francisco de Sande González

Incluso un viaje de miles de millas debe empezar con un solo paso.

Lao Tzu, filósofo chino

No importa cuán lento vayas mientras no te detengas.

Confucio

A mi familia

A Noemí

Agradecimientos

Quiero expresar mi más profundo agradecimiento a todas aquellas personas que, directa o indirectamente, han contribuido a hacer posible este trabajo.

Debo empezar agradeciendo a mi familia todo el apoyo prestado, ellos son los que realmente me han dado la oportunidad de haber llevado a cabo este proyecto y a ellos va dedicada: gracias de todo corazón a mis padres, mi hermana y mis abuelos. No tengo palabras para expresar mi gratitud a Noemí, su apoyo, ánimo, complicidad.... Con ella es con quién más he compartido el esfuerzo de este trabajo: esta tesis también te pertenece. Gracias al resto de mis familiares y a todos mis amigos, quienes también forman parte de mi familia.

Agradecer a mi director de tesis, Francisco de Sande González, su confianza depositada en mí, su valiosa labor de dirección y tutela. Gracias por el esfuerzo –espero que recompensado– de tantas y tantas horas invertidas delante de un monitor tratando temas relacionados con la investigación. Gracias más aún por los no relacionados.

Mis gracias también a todos los compañeros del Grupo de Paralelismo, del área de Lenguajes y Sistemas Informáticos y del Departamento de Estadística, Investigación Operativa y Computación, así como profesores de otros departamentos, por sus consejos y desinteresada ayuda prestada antes de comenzar mi investigación, durante y, sobre todo, en el difícil trabajo de sintetizar todo mi recorrido, plasmándolo en esta memoria.

Debo hacer una mención especial a todos los usuarios y administradores de los sistemas informáticos en los que hemos realizado ejecuciones: su colaboración y ayuda han sido fundamentales en este trabajo. También agradezco a mis compañeros de promoción, estudiantes, doctorandos y postdoctorandos con los que he colaborado y entablado amistad en el día a día, así como en congresos, conferencias y otros eventos: ustedes han sido los responsables de los recuerdos que enriquecen estos años. Animar a todos

aquellos que están terminando sus tesis doctorales y proyectos finales de carrera y agradecerles a todos sus contribuciones: espero que mis aportaciones les sean tan útiles como lo han sido las suyas para mí.

Agradecer a los distintos profesores con los que hemos realizado colaboraciones, en especial a Enrique Quintana y José Manuel Badía de la Universidad Jaime I de Castellón por su enorme ayuda y hospitalidad "*más allá del deber*". Gracias también a Judit Giménez de la Universidad Politécnica de Cataluña por la formación impartida en las herramientas `Paraver` y `Dimemás`.

A mis compañeros de trabajo darles las gracias por las facilidades y comprensión que me han brindado a la hora de cumplir con los trámites de la tesis, asistencia a eventos y demás ayuda que me han permitido compatibilizar esta tesis con mi trabajo, a ambos lados de la Transferencia.

Agradecer también a instituciones y organismos como el Centro de Investigaciones Energéticas, Medioambientales y Tecnológicas (CIEMAT), Centro Europeo de Paralelismo de Barcelona (CEPBA), Universidad Politécnica de Cataluña, Grupo de Computación Científica Paralela (PSCOM) de la Universidad Jaime I de Castellón, al Centro de Supercomputación de Edimburgo (EPCC), etc. que nos han permitido el acceso a sus equipos para la toma de resultados computacionales.

Gracias al programa *HPC-Europa* por brindarme la oportunidad de realizar una estancia en el EPCC, lo que supuso un gran impulso para mi investigación y también en el plano personal. Mi agradecimiento a los coordinadores de este programa en el EPCC, así como a los investigadores de esta institución, y un saludo a todos los doctorandos con los que coincidí aquellos meses en Edimburgo.

Para finalizar, es obligado agradecer también a las fuentes de financiación que han soportado parcialmente este trabajo ([45, 132, 3, 84]), permitiéndome asistir a distintos congresos y conferencias.

A los que he mencionado y a los que no ha sido posible por espacio, a todos, gracias.

Prólogo

Hasta ahora, la mejora en el rendimiento de los sistemas informáticos se basaba en gran medida en el aumento de la frecuencia y densidad de integración de transistores en los procesadores. Sin embargo, en la actualidad, este modelo es cada vez más complejo técnicamente y elevado su coste. Éste es uno de los motivos por los que el paralelismo está tomando una especial relevancia en el mundo de la computación, presentándose como una de las soluciones más viables y menos costosas que actualmente se conocen para seguir aumentando rendimiento y productividad.

La Computación de Altas Prestaciones (CAP) ha sabido absorber con éxito muchos de los problemas que hoy en día no son resolubles mediante sistemas uniprocesadores. Gracias a sistemas paralelos es posible dar respuesta a algoritmos que demandan una elevada cantidad de cómputo y donde el tiempo necesario para obtener los resultados es un factor crítico, como simulaciones meteorológicas, simulaciones en dinámica de partículas o fluidos, astrofísica, criptoanálisis, biotecnología, tratamiento de imagen y vídeo y un largo etcétera.

Tradicionalmente la CAP se ha encontrado con algunos problemas que dificultan su aplicación. El primero de estos problemas está relacionado con el hardware, teniéndose como primera limitación la de disponer de un sistema paralelo eficiente debido a su alto coste. Este problema puede considerarse hoy en día superado, ya que los actuales avances en la tecnología han abaratado los costes de este tipo de arquitecturas. Por ejemplo, el auge de los clusters de PCs pone las máquinas paralelas al alcance de la inmensa mayoría, a la vez que ofrece una excelente relación precio/rendimiento. El éxito de los sistemas paralelos basados en clusters queda patente en el `Top500` [49], un proyecto que desde 1993 recoge la clasificación y detalles de los 500 supercomputadores de propósito general más potentes del mundo, actualizando las listas dos o tres veces por año: en la última lista publicada en el momento de redacción de esta memoria (lista de junio de 2008), el 80 % de los computadores del `Top500` tenían una arquitectura basada en clusters.

A pesar de haberse salvado el principal escollo relacionado con el hardware, aún existen algunos aspectos que siguen ofreciendo dificultades. Uno de los más importantes es la diversidad de sistemas paralelos que existen en la actualidad: principalmente estos sistemas se clasifican en dos tipos de arquitecturas, de memoria compartida y de memoria distribuida, aunque también existen máquinas híbridas. Además, hay que considerar las diferentes topologías de comunicaciones, las máquinas con o sin acceso uniforme a memoria, etc.

La disparidad de hardware de sistemas paralelos no ha propiciado la uniformidad en el software. A pesar de que en cada arquitectura se han establecido estándares *de facto*, en estos momentos no existe un estándar único de programación paralela. Es indudable que hoy en día, la falta de lenguajes de alto nivel universalmente aceptados es el principal freno a la expansión generalizada de la CAP. Se presenta una dicotomía entre los usuarios que tienen necesidad de esta tecnología y los expertos que diseñan y desarrollan los lenguajes que permiten explotarla: habitualmente, quienes tienen la necesidad de cómputo intensivo no poseen los conocimientos necesarios para explotar las herramientas necesarias para el desarrollo de aplicaciones de CAP.

El futuro de la CAP y su popularización en los próximos años dependen, en gran medida, de que quienes investigamos en esta tecnología contribuyamos a cambiar este panorama. En este sentido será positivo cualquier esfuerzo destinado a reducir la distancia entre usuarios y herramientas, incrementando la simplicidad y el nivel de abstracción de los lenguajes.

Se desarrollan en la actualidad diversos proyectos que buscan la mejora de prestaciones y uso de los sistemas paralelos. Entre ellos destaca el desarrollado por la agencia americana DARPA (**D**efense **A**dvanced **R**esearch **P**rojects **A**gency) [43], que ha creado un ambicioso proyecto cuyo principal objetivo es incrementar la productividad de los supercomputadores para el año 2010. Se trata del programa HPCS (**H**igh **P**erformance **C**omputing **S**ystems) [104] y dentro de su ámbito investigan las principales universidades y empresas del sector.

En el Grupo de Computación Paralela de la Universidad de La Laguna [127], desde el comienzo de nuestra actividad hemos mantenido una línea de investigación centrada en el diseño, implementación y evaluación de lenguajes de alto nivel para CAP [98, 44]. El presente trabajo representa el más reciente hito en esta línea de trabajo.

En la actualidad, `MPI` (Message Passing Interface) [109], es la herramienta

más utilizada para desarrollar aplicaciones paralelas. Se trata de una librería de rutinas cuyas más claras ventajas son su portabilidad (está disponible para arquitecturas de memoria compartida y distribuida) y el alto rendimiento de las aplicaciones desarrolladas. Si se analizan las debilidades de `MPI` como herramienta de desarrollo de aplicaciones paralelas, encontramos que la más relevante es la baja productividad debido a que el control explícito de la semántica paralela queda bajo la responsabilidad del programador. Desarrollar una aplicación paralela usando `MPI` requiere un considerable esfuerzo y un alto grado de especialización. Podríamos decir que `MPI` representa el ensamblador de los lenguajes para CAP: permite conseguir muy altas prestaciones pero a costa de un gran esfuerzo de desarrollo.

Como alternativa a `MPI`, en la última década `OpenMP` [123] ha surgido como el estándar *de facto* para la programación en entornos de memoria compartida. La API de `OpenMP` se basa en un pequeño número de directivas junto con algunas rutinas de librería y variables de entorno. La rápida expansión y el éxito de `OpenMP` se deben en gran medida a su simplicidad y a su alto grado de programabilidad en comparación con otras alternativas actuales. Habitualmente desarrollar una primera versión paralela de una aplicación es fácil puesto que la versión secuencial no precisa cambios esenciales. Como contrapartida, también es cierto que obtener un alto rendimiento de un programa `OpenMP` requiere esfuerzo especializado en sintonización de la aplicación. La creciente popularidad de los clusters de PCs, motivada por su buena relación precio/prestaciones, deja claro el interés de cualquier esfuerzo para portar códigos `OpenMP` a arquitecturas de memoria distribuida, con una pérdida de eficiencia razonable. Indudablemente, la falta de portabilidad de `OpenMP` al mundo de las arquitecturas de memoria distribuida es su principal carencia.

El presente trabajo se enmarca en el contexto del desarrollo de nuevos lenguajes de programación para CAP. Nuestro trabajo ha sido guiado por el objetivo de diseñar un lenguaje en el que la programabilidad tiene un papel central.

Las aportaciones que se realizan en esta tesis pueden resumirse como sigue:

- Presentamos `llc` (**L**a **L**aguna **C**) [56], el lenguaje que hemos diseñado para extender `OpenMP` a sistemas de memoria distribuida. `llc` pretende aunar las bondades de `MPI` y `OpenMP` como estándares dominantes en el momento actual, para conseguir el alto nivel y la facilidad de programación de `OpenMP` combinadas con la eficiencia y portabilidad de `MPI`.

- `llCoMP`, el compilador de `llc`, es la segunda aportación de nuestro trabajo. La adopción del Modelo de Computación Colectiva (OTOSP) [44] junto con la aproximación consistente en hacer que `llCoMP`, genere código para las comunicaciones sobre la librería `MPI` nos permite simplificar la implementación del compilador.

- Para validar nuestra aproximación, hemos implementado usando `llc` un amplio abanico de algoritmos. La eficiencia de la traducción que realiza `llCoMP` se compara con el código secuencial, así como con versiones `OpenMP` y/o `MPI` del correspondiente algoritmo. Esta tarea de validación experimental es la tercera aportación de la tesis.

- La cuarta aportación de este trabajo ha sido el diseño e implementación de `OmpSCR`, un repositorio de código fuente `OpenMP` a través del cual los usuarios pueden comparar y evaluar diferentes implementaciones en este lenguaje.

Esta memoria está estructurada en cinco capítulos. En el primer capítulo realizamos un estudio, que no pretende ser exhaustivo, de lenguajes actuales utilizados en el desarrollo de aplicaciones paralelas. Ponemos de manifiesto las fortalezas y debilidades de cada uno de los casos analizados.

El segundo capítulo está dedicado a presentar `llc`. Se hace un recorrido por la sintaxis y semántica del lenguaje, explicando los constructos que utiliza para expresar paralelismo.

En el tercer capítulo se estudia el compilador `llCoMP`. Es importante mencionar que el diseño de `llCoMP` no ha sido el aspecto central de este trabajo, y por ello el esfuerzo que se ha dedicado al compilador ha sido limitado, poniendo el énfasis en probar nuestras ideas mediante la implementación de diferentes algoritmos y midiendo su rendimiento en un amplio espectro de sistemas.

El cuarto capítulo recoge esta tarea de experimentación. Se presenta un buen número de algoritmos que han sido implementados con `llc` y en muchos casos también usando `MPI` y/o `OpenMP` y comparamos el rendimiento de las diferentes aproximaciones en varias arquitecturas.

La memoria finaliza con la presentación en el quinto capítulo de las conclusiones de este trabajo, así como de futuras líneas de investigación.

Índice general

Índice de figuras

Índice de tablas

Índice de listados

Capítulo 1

Lenguajes modernos para computación de altas prestaciones

1.1. Introducción

En este capítulo realizaremos un recorrido por el estado del arte de la programación paralela. Se presentarán los lenguajes que han sido el estándar de la programación paralela en el pasado y que han servido de base para los lenguajes actuales [7], se estudiarán éstos y cómo han evolucionado y se mostrará el estado de las investigaciones actuales para desarrollar los lenguajes de programación paralela que establecerán el estándar en los próximos años, muchas de ellas realizadas bajo el ámbito del programa HPCS [104].

El conjunto de lenguajes, librerías y herramientas paralelas es lo suficientemente amplio como para que haya sido necesario acotarlo en esta memoria, presentando en este capítulo sólo aquellos que poseen una mayor relevancia. Además de éstos, existe un grupo de lenguajes que, a pesar de no haber tenido una especial trascendencia a nivel global, sí han repercutido e influenciado directamente nuestro trabajo. Algunos de ellos serán mencionados en este capítulo, mientras que otros han sido omitidos por cuestiones de espacio ([80, 78, 136, 14, 85]).

Para no extender en exceso este capítulo, hemos restringido a sólo unos párrafos la información que se presenta en cada caso. Para algunos de los lenguajes con los que hemos trabajado directamente también hemos

añadido observaciones y comentarios obtenidos de la propia experiencia. Finalmente, hemos incluido fragmentos de códigos con el objetivo de ilustrar las explicaciones con ejemplos de programas codificados en cada lenguaje. El lector interesado en ampliar conocimientos sobre alguno de los lenguajes que aquí se resumen puede consultar las correspondientes referencias de la bibliografía.

El orden que hemos seguido para presentar los lenguajes ha sido una combinación entre un orden cronológico y de relevancia en nuestro trabajo, agrupándolos por características comunes. De este modo, los dos primeros lenguajes que se tratarán serán `OpenMP` y `MPI` debido a su impacto en nuestro desarrollo. Seguidamente se estudiarán las librerías de comunicaciones (dentro de la que se encuentra `MPI`) y los lenguajes PGAS. Finalmente, se presentarán una serie de lenguajes que no pertenecen a la clasificación anterior, para terminar con varios de los trabajos más recientes, enmarcados en el programa HPCS.

1.2. `OpenMP`

`OpenMP` (**Open** **M**ulti-**P**rocessing) [42, 28, 17] es una API (**A**pplication **P**rogram **I**nterface) que nace del trabajo conjunto de importantes fabricantes de hardware y software, ofreciendo al usuario un modelo de programación de paralelismo explícito. Esta API está compuesta por un conjunto de directivas de compilador que el usuario utiliza para especificar qué regiones de código van a ser paralelizadas, y una librería de rutinas. El usuario también cuenta con un limitado número de variables de entorno que permiten especificar, entre otros aspectos, las condiciones que definirán la ejecución de la aplicación paralela. `OpenMP` se especifica para `C/C++` y `Fortran`.

La implementación de `OpenMP` utiliza *multithreading* para obtener el paralelismo, mediante un modelo `fork-join`. De este modo, al encontrarse una directiva paralela que así lo indique, el *thread maestro* se divide en un número determinado de *threads esclavos* que se ejecutan concurrentemente y las tareas se distribuyen entre ellos. Dependiendo de los parámetros indicados, carga de la máquina y otros factores, el entorno de ejecución asignará uno o varios de los *thread esclavos* a los distintos procesadores disponibles.

`OpenMP` se define específicamente para trabajar sobre máquinas de memoria compartida y, de hecho, espera convertirse en el estándar de programación paralela en este tipo de sistemas. En `OpenMP` todos los threads

acceden a la misma memoria, aunque es posible gestionar estos accesos y generar espacios de memoria privada, entre otras operaciones. De ello se encargan algunas de sus directivas, que se ven complementadas por las *cláusulas*, con diversos usos dependiendo de la directiva a la que acompañen. En general, estas cláusulas se usan para modificar el carácter de las variables (privadas o compartidas), para llevar a cabo operaciones de sincronización, inicialización, copia de datos, reducción, control de flujo, etc.

El modelo de OpenMP se destaca por ofrecer a los usuarios una interfaz simple y sencilla para desarrollar aplicaciones paralelas en un amplio rango de arquitecturas de memoria compartida. Esta simplicidad se debe, en gran medida, al uso de las directivas de compilador que permiten partir de un código secuencial y paralelizarlo sin necesidad de realizar grandes cambios: basta insertar las directivas para marcar aquellas regiones susceptibles de ser paralelizadas, lo cual también reduce considerablemente el tiempo de depuración al disminuir las posibilidades de introducir nuevos errores en el código, si se parte de un programa secuencial correcto.

De este modo, no es necesario lidiar con paso de mensajes ni con las estructuras de datos, función realizada automáticamente por las directivas. Además de la sencillez de su modelo de programación, también sobresale la facilidad y reducido tiempo de aprendizaje para aquellos usuarios neófitos en la programación paralela: estos sólo deben conocer el limitado conjunto de directivas y cláusulas de OpenMP, así como algunos conceptos básicos de paralelismo como threads, memoria compartida y distribuida, etc.

Otra ventaja añadida de este enfoque es que la paralelización puede ser llevada a cabo de forma incremental, marcando o desmarcando regiones paralelas con directivas según se requiera. Es más, el hecho de añadir directivas de OpenMP no rompe la estructura del programa secuencial, que sigue siendo válido para compiladores secuenciales, que ignoran las directivas de OpenMP tratándolas como comentarios.

Sin embargo, OpenMP también presenta una serie de desventajas. La más importante de ellas es que en la actualidad OpenMP ha sido diseñado específicamente para ser ejecutado sobre máquinas de memoria compartida. El hecho de que OpenMP no esté concebido para sistemas de memoria distribuida es un factor limitante que reduce drásticamente el conjunto de máquinas objetivo sobre la que se pueden ejecutar las aplicaciones, sobre todo teniendo en cuenta el auge actual de supercomputadoras de memoria distribuida, como los clusters [49]. Por otro lado, las aplicaciones desarrolladas con OpenMP se ven afectadas por problemas de escalabilidad, derivados de las limitaciones del ancho de banda de las máquinas de memoria

compartida.

OpenMP también presenta otros problemas, como la dificultad en ajustar los parámetros para obtener un alto rendimiento, escasez de compiladores que implementen un paralelismo multinivel de forma eficiente (a pesar de que el estándar soporta este tipo de paralelismo), diferencias en la implementación (e incluso sintaxis) en versiones de diferentes fabricantes, falta de garantía sobre la eficiencia del uso de la memoria compartida (actualmente no hay constructos para gestionar la localidad de datos), etc.

El listado 1.1 muestra un código secuencial que calcula una aproximación del número π mediante un algoritmo iterativo de aproximaciones rectangulares. Para obtener un código paralelo OpenMP a partir de este código secuencial sólo es necesario añadir las directivas de compilador donde sea preciso.

```
1  w = 1.0 / N;
2  pi_seq = 0.0;
3  for (i = 0; i < N; i++) {
4   local = (i + 0.5)*w;
5   pi_seq = pi_seq + 4.0/(1.0 + local*local);
6  }
7  pi_seq *= w;
```

Listado 1.1: Ejemplo del cálculo de π mediante un código secuencial

El listado 1.2 muestra el resultado de la paralelización de este código mediante el uso de directivas de OpenMP. En la línea 3 se abre una región paralela mediante la directiva `parallel` con dos variables privadas a cada thread (cláusula `private`). Luego, en la línea 5 se utiliza la directiva `single` para indicar que el siguiente tramo de código (inicialización en línea 6) sólo sea ejecutado por un thread. Finalmente, en la línea 8, se especifica que el bucle `for` de la línea 9 debe realizarse en paralelo (directiva `omp for`) y que, al finalizar la ejecución, se realizará una operación de reducción del tipo suma sobre la variable `pi_omp` (cláusula `reduction`). Comparando este código con el secuencial, queda patente la simplicidad con la que, en general, se pueden obtener códigos paralelos OpenMP a partir de aplicaciones secuenciales.

OpenMP tiene una especial repercusión sobre nuestro trabajo: las características de su modelo de programación y, sobre todo, su simplicidad, han servido de punto de partida para nuestra aproximación. Éste, básicamente, se basa en el diseño e implementación de un lenguaje y modelo

```
1   w = 1.0 / N;

3 #pragma omp parallel private(i, local)
4   {
5 #pragma omp single
6     pi_omp = 0.0;

8 #pragma omp for reduction (+: pi_omp)
9     for (i = 0; i < N; i++) {
10      local = (i + 0.5)*w;
11      pi_omp = pi_omp + 4.0/(1.0 + local*local);
12    }
13  }
14  pi_omp *= w;
```

Listado 1.2: Ejemplo del cálculo de π mediante un código `OpenMP`

de programación similar a `OpenMP` que solucione algunos de las limitaciones de este lenguaje, como, entre otras, la portabilidad a sistemas de memoria distribuida. Algunos aspectos de `OpenMP` que han tenido relevancia en nuestro trabajo serán tratados con más detalle en el capítulo 2 dedicado a nuestro lenguaje.

1.3. Librerías de Comunicación

El paradigma de librería de paso de mensajes se estableció en los años 90 del siglo pasado, cuando la comunidad de la CAP entendió que la aproximación en boga hasta ese momento, que consistía en extender lenguajes ya existentes con directivas que el compilador analizaba, no era suficientemente rica como para describir los algoritmos emergentes en ese momento.

En el modelo de paso de mensajes (MPI [109, 140], PVM [73, 74] y P4 [19] son ejemplos significativos), el programador controla directamente el flujo de operaciones y de datos en su programa. La librería permite indicar a cada procesador qué ha de hacer y suministra un mecanismo de comunicación y cooperación entre procesos. La funcionalidad de la librería se define al margen de un lenguaje de programación específico, y la implementación para un determinado lenguaje se suministra como una API.

Curiosamente, en la actualidad se está volviendo a adoptar la

aproximación de incorporar paralelismo en lenguajes ya existentes (Co-array Fortran, UPC, Titanium) [7]. Estos nuevos lenguajes incorporan un número mínimo de constructos a lenguajes ya existentes (Fortran95, C y Java respectivamente), para incorporar soporte para paralelismo.

1.3.1. MPI

MPI (**M**essage **P**assing **I**nterface) [109, 140] es un estándar de librería de paso de mensajes, desarrollado por un comité que abarca un amplio conjunto de fabricantes, universidades, laboratorios gubernamentales, científicos, usuarios, etc..

MPI, en sí mismo, no es un lenguaje, sino una librería, mientras que el estándar refleja la especificación de la semántica y protocolos, pero no su implementación. Entre otras razones, MPI surgió hace más de una década para dar respuesta a la crisis del software de programación paralela, debida a que cada fabricante ofrecía su propia interfaz, diferente de las restantes, sin que ninguna de ellas lograra la suficiente masa crítica de usuarios para convertirse en un estándar. Por otro lado, el uso de librerías permitió a la comunidad distribuir, reutilizar y compartir sus códigos.

El comité que desarrolla MPI destina un gran esfuerzo con el fin de lograr estándares bien especificados. Tal es así que la primera versión 1.0 de MPI (5 de mayo de 1994) fue fruto de la colaboración y esfuerzo de más de 40 organizaciones durante dos años de trabajo intensivo. Desde ese entonces hasta nuestros días tan sólo se han publicado tres nuevas versiones MPI (1.1, 1.2 y 2.0). Este reducido número de versiones en casi 15 años acredita la solidez de sus especificaciones.

La fortaleza del estándar MPI, junto con otras de sus características, como la libre disponibilidad de sus implementaciones, el elevado número de las mismas para diferentes arquitecturas, la portabilidad, aplicaciones con un alto rendimiento, escalabilidad, etc., han convertido a MPI en un estándar *de facto* en programación de entornos distribuidos, prácticamente desplazando en computación técnica al resto de sistemas de paso de mensajes.

MPI se estructura en dos fases. La primera de ellas, el estándar MPI-1[1], consiste en un paso de mensajes tradicional, mientras que MPI-2 amplía la primera fase, extendiendo el modelo de paso de mensajes (E/S paralela, operaciones en memoria remota, gestión de procesos dinámicos,

[1] en esta memoria, salvo que se indique explícitamente lo contrario, el término MPI hace referencia al estándar MPI-1

etc.), así como añade nuevas funcionalidades, como interfaces externas, interoperabilidad de lenguajes, interacción con threads, etc. En la actualidad, la mayoría de fabricantes de sistemas paralelos ya incorporan `MPI-2` en sus máquinas.

En el modelo de paso de mensajes en el que se basa `MPI`, los programas paralelos consisten en procesos que cooperan, cada uno con su propia memoria privada. Estos procesos envían datos e información unos a otros como mensajes. Los mensajes pueden ser recibidos en cualquier orden, por lo que para poder gestionarlos y ordenarlos, a cada uno de ellos se le asigna una etiqueta. Para el envío de mensajes `MPI` utiliza una serie de *comunicadores* que permiten crear grupos arbitrarios de procesos, teniendo cada uno de ellos un identificador único por cada grupo al que pertenezca.

Para la gestión de los comunicadores y las propias comunicaciones, `MPI` pone disposición del usuario un amplio conjunto de funciones. Las comunicaciones deben ser indicadas explícitamente por el usuario, realizando llamadas a funciones que implementan comunicaciones punto a punto o colectivas, bloqueantes o no bloqueantes, etc. En ellas, el usuario especifica el origen y el destino mediante el uso de los comunicadores e identificadores de procesos, la cantidad de datos que se van a transmitir y el tipo de los mismos, teniendo que usar para ello identificadores de tipos propios de `MPI`. Para ello, `MPI` dispone por defecto de los tipos de datos predefinidos comunes, pero si el usuario hace uso de tipos complejos deberá usar funciones para definirlos, o bien comunicar estos datos como una ristra de bytes, teniendo que gestionar los buffers de comunicación en caso de que se requiera, y teniendo en cuenta, además, que los mensajes pueden llegar en cualquier orden.

En definitiva, con `MPI` el usuario debe codificar todo el paralelismo de su aplicación, desde iniciar el entorno paralelo, crear los grupos de procesos, distribuir los datos y tareas entre ellos, especificar todas las comunicaciones, etc., hasta la finalización del entorno paralelo. Esto requiere que para crear programas paralelos en `MPI` se precisen profundos conocimientos de paralelismo y, en algunos casos, de la arquitectura paralela. Todo esto origina que los programas paralelos que se obtienen en `MPI`, incluso para los casos más simples, raramente se asemejan a su homólogo secuencial.

Estos requisitos hacen de `MPI` una aproximación de muy bajo nivel, llegando a ser considerado por algunos autores como "*el ensamblador de las aplicaciones paralelas*" [5, 116]: con `MPI` es posible obtener programas paralelos tan eficientes como optimizado esté el código creado, pero a costa de poseer conocimientos avanzados sobre paralelismo y la propia librería de `MPI` y tras un largo, y generalmente tedioso, proceso de programación y

depuración.

De este modo, MPI se convierte en una de las mejores alternativas para la creación de programas paralelos, tanto para sistemas de memoria distribuida como compartida, con el inconveniente de ser viable sólo para usuarios con amplios conocimientos sobre programación y paralelismo.

Comparando las características de MPI con las de OpenMP, queda patente que sus ventajas se complementan. Por un lado, OpenMP aporta un modelo de programación sencillo, tanto durante el proceso de aprendizaje como en su uso para generar programas paralelos. Por otro lado, MPI permite crear programas portables, con un alto rendimiento y escalabilidad, e independientes de la arquitectura de memoria compartida o distribuida en la que vayan a ser ejecutados.

Nuestro trabajo se ha centrado en desarrollar un lenguaje y su compilador que implementen las ventajas de ambas opciones. Mientras que el usuario trabaja con un lenguaje similar a OpenMP, el compilador es capaz de generar el código MPI eficiente que implemente el paralelismo especificado por el usuario. Es por este motivo que MPI, al igual que OpenMP, tiene gran relevancia en el trabajo que hemos realizado y en capítulos posteriores será abordado nuevamente.

```
1   MPI_Init (&argc, &argv);
2   MPI_Comm_size(MPI_COMM_WORLD, &MPI_NUMPROCESSORS);
3   MPI_Comm_rank(MPI_COMM_WORLD, &MPI_NAME);

5   w = 1.0 / N;
6   pi_mpi = 0.0;
7   for(i = MPI_NAME; i < N; i += MPI_NUMPROCESSORS) {
8     local = (i + 0.5) * w;
9     pi_mpi = pi_mpi + 4.0/(1.0 + local*local);
10  }

12  MPI_Allreduce(&pi_mpi, &gpi_mpi, 1, MPI_DOUBLE, MPI_SUM,
      MPI_COMM_WORLD);
13  MPI_Barrier(MPI_COMM_WORLD);
14  pi_mpi = gpi_mpi * w;

16  MPI_Finalize ();
```

Listado 1.3: Ejemplo del cálculo de π mediante un código MPI

El listado 1.3 muestra el código MPI que implementa el cálculo de la aproximación del número π. En este ejemplo el usuario ha tenido que especificar explícitamente todo el paralelismo, desde la inicialización del entorno paralelo (MPI_Init(), línea 1), la obtención del tamaño e identificador dentro del comunicador actual (MPI_Comm_size() y MPI_Comm_rank(), líneas 2 y 3), la distribución del paralelismo mediante la división del espacio de iteraciones del bucle paralelo (bucle for en línea 7) y las posteriores operaciones de comunicación (MPI_Allreduce(), línea 12) como de sincronización (MPI_Barrier() en la línea 13), para terminar finalizando el entorno paralelo (MPI_Finalize(), línea 16).

A pesar de la sencillez del ejemplo podemos constatar la dificultad de generar este código si lo comparamos con el código paralelo equivalente de OpenMP que se recoge en el listado 1.2 (página 5). También queda patente la ruptura de la estructura secuencial del programa, si lo comparamos con el listado 1.1 (página 4).

1.3.2. PVM

Los orígenes de PVM (**P**arallel **V**irtual **M**achine) [73, 74] datan de 1989. La necesidad de definir un marco para una investigación sobre computación en sistemas heterogéneos dio lugar al concepto de *máquina paralela virtual*, que da nombre al lenguaje. A partir de su segunda versión (PVM 2.0, en 1991), el uso de esta utilidad creció rápidamente entre los científicos de todo el mundo dedicados a investigación computacional.

La idea central de PVM fue una "*máquina virtual*", un conjunto de sistemas heterogéneos conectados mediante una red y que se presentan al usuario como una única máquina paralela. Este concepto de máquina virtual marca todo el diseño de PVM y sienta la base para la heterogeneidad, portabilidad y encapsulación de sus funciones.

Las tareas que se ejecutan en esta máquina virtual intercambian los datos a través de *paso de mensajes* simples. En un principio, PVM fue diseñado de forma que su interfaz fuera simple y fácil de entender y usar, primando la portabilidad sobre el rendimiento. A medida que el uso de PVM fue creciendo, nuevas versiones fueron apareciendo para dar respuestas a las nuevas necesidades.

PVM se presenta como la integración de un conjunto de utilidades y librerías que emulan un entorno de computación concurrente flexible, de propósito general, y heterogéneo para computadoras interconectadas de

diversas arquitecturas. Las principales características de PVM residen en la capacidad que tiene el usuario de poder configurar el conjunto de máquinas en las que se ejecutarán las aplicaciones, varias formas de ver y acceder al hardware, computación basada en procesos, modelo de paso de mensajes explícitos, soporte para sistemas heterogéneos y sistemas multiprocesador.

El sistema PVM se compone de dos partes. Por un lado, existe un demonio (*daemon*) que reside en todas las computadoras que forman la máquina virtual. La segunda parte del sistema es la librería de PVM, que contiene el conjunto de primitivas necesarias para la ejecución de las tareas de una aplicación, incluyendo aquellas de paso de mensajes, creación, manejo y coordinación de tareas y la gestión de la máquina virtual.

PVM fue el estándar *de facto* en computación distribuida hasta que posteriormente MPI ocupó este papel. El hecho de que tanto PVM como MPI son especificaciones para librerías que pueden ser usadas para computación paralela, con grandes similitudes entre ellas, creó un gran desconcierto entre los programadores y existen diversas publicaciones dedicadas a comparar ambos lenguajes y discutir sus afinidades y diferencias [74]. Incluso existe un proyecto que trabaja la idea de unir las características de estos dos lenguajes para crear uno único, denominado PVMPI, creando así un un entorno que permita sacar partida de las características de una máquina virtual como la de PVM con el paso de mensajes de MPI. [66].

PVM está disponible para lenguajes como C, C++ y Fortran y la última versión de PVM es la 3.4.5 en el momento de redacción de esta memoria. PVM también dispone de una consola gráfica y monitor, denominado XPVM.

El listado 1.4 recoge un ejemplo que hemos implementado en PVM3. Se trata del bucle principal del procesador maestro que calcula la suma de los valores guardados en los nodos esclavos siguiendo una topología de hipercubo de DIM dimensiones. Como puede observarse en este ejemplo, la forma de crear un código paralelo consiste en realizar llamadas a la librería de PVM (funciones que típicamente empiezan por pvm_...), siguiendo una filosofía similar a la de MPI.

1.3.3. SHMEM

El modelo de programación de SHMEM [10] extiende las capacidades de la programación paralela explícita de MPI. Este modelo permite que un procesador pueda colocar (*put*) y obtener (*get*) datos de la memoria de otro procesador, sin que el código de este procesador se vea afectado.

```
1  for (i = 0; i < DIM; i++) {
2    /* Prepara el mensaje para el vecino determinado  */
3    /* según la dimensión con la que se quiera comunicar */
4    rc = pvm_initsend(PvmDataDefault);
5    rc = pvm_pkint(&suma,1,1);
6    rc = pvm_send(tids[PARTNER(name,i)], msgtype);

8    /* Recibe la suma parcial realizada por su vecino */
9    rc = pvm_recv (tids[PARTNER(name,i)], msgtype);
10   rc = pvm_upkint (&suma_recv,1,1);

12   /* Realiza la nueva suma que en la próxima iteración */
13   /* será enviada al vecino de dimensión superior */
14   suma += suma_recv;
15 }
```

Listado 1.4: Ejemplo de un código `PVM`

La librería de `SHMEM` dispone de una de las comunicaciones interprocesadores más rápidas de las industria, usando técnicas de *data passing* o comunicaciones unidireccionales (*one-sided communication*). Esto se debe a la eliminación de parte de la sincronización y *buffering* que necesita el paso de mensajes bidireccional (*two-sided*).

Este estilo de comunicación unidireccional es sustancialmente más fácil de usar, ya que los programadores no tienen que escribir código para conocer qué comunicaciones están llevando a cabo otras instancias del programa, evitando así las sincronizaciones que normalmente se requieren para determinar cuándo los datos remotos están listos para ser leídos o escritos.

Esta librería también dispone de la capacidad para asignar punteros globales que permiten acceder directamente a datos de otros procesos cooperantes, mediante operaciones `load`/`store` de comunicación o sincronización. Además, la librería `SHMEM` incluye varias rutinas altamente optimizadas para operaciones colectivas, como reducciones globales.

Entre las características de la librería `SHMEM` podemos citar:

- su interfaz simple de memoria compartida
- operaciones `get` y `put` unidireccionales
- menor sobrecarga y mayor rendimiento que las operaciones

tradicionales bidireccionales

- interfaz más natural en muchas aplicaciones
- operaciones colectivas altamente eficientes
- capacidad de punteros remotos globales
- implementaciones optimizadas
- etc.

Existe una versión de `SHMEM`, denominada `GPSHMEM` (***G**eneralized **P**ortable SHMEM*) [126], que ha sido diseñada para dotar de portabilidad a la interfaz en un amplio rango de plataformas.

El listado 1.5 recoge un ejemplo de una aplicación `SHMEM`. En concreto, se trata de una porción de código que implementa una versión multiproceso del comando `ping` de `UNIX`, extendiendo este comando para trabajar en una red de múltiples equipos ejecutando procesos paralelos sobre un número dado de procesadores. Los procesos forman pares y cada proceso realiza un *ping* a su par un cierto número de veces, para finalizar imprimiendo estadísticas de tiempo (para simplificar el ejemplo, hemos eliminado el código relacionado con la toma de tiempos).

Como ocurre con otras librerías, el código paralelo comienza con la llamada a la función de inicialización de la librería (`shmem_init()`, línea 4). El siguiente paso es obtener el identificador del proceso dentro del grupo y el número de procesos que lo componen, llamando a las respectivas funciones (`my_pe()` y `my_pes()`, líneas 5 y 6). Posteriormente se realiza una sincronización (función `shmem_barrier_all()`, línea 8) para asegurar que todos los procesos se han inicializado correctamente . Esta sincronización se repite en la línea 15 para asegurar que todos los procesos están listos para enviar y recibir mensajes, y también en la línea 32, al finalizar el bucle, para garantizar que todos los procesos lo han completado, antes de concluir la ejecución. Las llamadas a la función `shmem_put()` (líneas 23 y 28) permiten a un proceso copiar datos en la variable remota de su par, indicando para ello el buffer remoto, el local, el número de palabras a copiar y el identificador del proceso par. A su vez, la función `shmem_wait()` (líneas 19 y 24) tiene el objetivo de bloquear el proceso hasta que su proceso par haya terminado de modificar los valores del buffer.

```
1  int main(int argc, char *argv[]) {
2
3    ...
4    shmem_init();
5    proc = my_pe();
6    nproc = num_pes();
7    ...
8    shmem_barrier_all();
9    ...
10
11   for (nwords = minWords; nwords <= maxWords;
12        nwords = incWords ? nwords + incWords : nwords ?
13        2 * nwords : 1)  {
14     r = reps;
15     shmem_barrier_all();
16     if (peer < nproc) {
17       if (proc & 1) {
18         r--;
19         shmem_wait(&rbuf[nwords-1], 0);
20         rbuf[nwords-1] = 0;
21       }
22       while (r--- > 0) {
23         shmem_put(rbuf, tbuf, nwords, peer);
24         shmem_wait(&rbuf[nwords-1], 0);
25         rbuf[nwords-1] = 0;
26       }
27       if (proc & 1)
28         shmem_put(rbuf, tbuf, nwords, peer);
29       ...
30     }
31   }
32   shmem_barrier_all();
33   exit (0);
34 }
```

Listado 1.5: Ejemplo de un código SHMEM

1.3.4. ARMCI

`ARMCI` (***A**ggregate **R**emote **M**emory **C**opy **I**nterface*) [114] es una librería de comunicación de acceso remoto a memoria (*RMA*) de alto rendimiento, propósito general, eficiente y ampliamente portable. El desarrollo de `ARMCI` se debe a la necesidad de ofrecer soporte al modelo de comunicación de espacio de direcciones global en el contexto de estructuras de datos distribuidas (tanto regulares como irregulares), librerías de comunicación y compiladores.

Sus principales características son:

- operaciones de transferencia de datos bloqueantes y no bloqueantes
- fusión de transferencias de datos pequeñas en mensajes más grandes para reducir la sensibilidad a la latencia de la red
- operaciones atómicas y de sincronización
- operaciones de gestión de memoria
- grupos de procesadores
- llamadas a procedimientos globales
- modelo de consistencia de memoria débil
- etc.

`ARMCI` está optimizada para transferencia de datos contiguos y no contiguos. Una operación `get` transfiere datos desde la memoria de un proceso remoto (fuente) hacia la memoria local del proceso que realizó la llamada (destino). Una operación `put` realiza el procedimiento inverso, transfiriendo datos desde la memoria local del proceso llamante (fuente) a la memoria del proceso remoto (destino). La API *no-bloqueante* es derivada de la interfaz bloqueante, añadiendo un argumento que identifica la instancia de la petición no-bloqueante. Esta librería incluye un pequeño grupo de operaciones colectivas de paso de mensajes, como `broadcast`, `reduce`, `allreduce`, `barrier`, ..., así como un conjunto de operaciones atómicas y de exclusión mutua.

`ARMCI` dispone de mecanismos para reducir la latencia, como la *fusión de mensajes cortos*. Para mejorar la tolerancia a la latencia, las peticiones se agrupan, de modo que múltiples peticiones de transferencias no bloqueantes (`put`/`get`) pueden ser compactadas en una sola operación de transferencia

de datos, para mejorar así la tasa de transferencia de datos. Cada una de esas peticiones puede ser de diferente tamaño e independiente del tipo de datos, pudiendo cada operación de transferencia ser incluso independiente del tipo de operación `put/get`. Existen dos tipos de agrupación, el *explícito*, donde múltiples peticiones son combinadas a partir de las especificaciones del usuario, o bien el *implícito*, donde la combinación de peticiones individuales es realizada por `ARMCI`.

Otras características de `ARMCI` son:

- *transferencia de datos originadas por registros* que permiten operaciones de transferencia registro-memoria (`value_put/value_get`), donde el destino es un registro del proceso local y el destino es la memoria del proceso remoto, evitando así la sobrecarga de la capa del buffer de gestión (subsistema local de memoria)
- *operaciones atómicas*, en concreto `accumulate` y `read-modify_write`
- *exclusiones mutuas (mutex) y bloqueos*, ya que `ARMCI` soporta operaciones mutex distribuidas, pudiendo el usuario crear un conjunto de mutexs asociados a un proceso específico y usar las operaciones de bloqueo `ARMCI_Lock()/ARMCI_Unlock()` en mutex individuales de ese conjunto
- *funciones de asignación de memoria* que `ARMCI` ofrece y que, por razones de rendimiento, deben usarse para asignar los datos remotos (`ARMCI_Malloc()`) o memoria local (`ARMCI_Malloc_local()`)

`ARMCI` depende de una librería de paso de mensajes (como `MPI`) para la creación de los procesos y la gestión del entorno de ejecución. Sin embargo, por diseño, `ARMCI` es imparcial con respecto a la elección de la librería de paso de mensajes realizada en la aplicación del usuario. `ARMCI` se presenta como un sistema autónomo que puede usarse para dar soporte a librerías o aplicaciones que usan paso de mensajes, como `MPI` (con quien es compatible) u otras (para algunas plataformas también es posible usar `PVM` y `TCGMSG`). El usuario puede incluso mezclar llamadas a la librería de paso de mensajes entre las llamadas a `ARMCI`.

`ARMCI` explota las interfaces de comunicación de red nativas y los recursos del sistema (como la memoria compartida) para lograr el mejor rendimiento posible de los accesos a memoria remotos/comunicaciones unilaterales. `ARMCI` también hace uso de los protocolos de alto rendimiento en clusters. De esta forma, existen versiones de `ARMCI` optimizadas para *Myrinet*, *Quadrics*,

Giganet (VIA) e *Infiniband*. Además, `ARMCI` está disponible para un amplio conjunto de plataformas, entre las que se pueden citar IBM BlueGene/L y SPs; diversos modelos de Cray, servidores y estaciones de trabajo Unix (Sun, SGI, HP, IBM, ...) y linux; clusters de servidores y estaciones de trabajo UNIX y Windows NT, etc. Existen diversas librerías y compiladores que usan `ARMCI`, tales como la librería de `Global Array`, `GPSHMEM`, compilador de `Co-Array Fortran`, etc.

El listado 1.6 ilustra un ejemplo de un código que, mediante el uso de definiciones, permite la creación de aplicaciones `ARMCI` independientemente de la librería de paso de mensajes que se utilice. Este ejemplo muestra las definiciones para barreras (`MP_BARRIER()`), inicialización del entorno de la librería (`MP_INIT()`), obtención del identificador de proceso (`MP_MYID()`) y de número de procesos del grupo (`MP_PROC()`) y finalización del entorno de la librería (`MP_FINALIZE()`). Las definiciones se llevan a cabo para cada uno de las siguientes librerías de paso de mensajes: `PVM` (líneas 3 a 15), `TCGMSG` (líneas 16 a 23) y `MPI` (líneas 25 a 30).

El listado 1.7 usa las definiciones presentadas en el listado 1.6 para mostrar una plantilla de una aplicación típica en `ARMCI`. En las líneas 7 a 9 se realiza la inicialización del entorno de la librería de paso de mensajes que se haya escogido, obteniendo también el identificador y número de procesos en el grupo actual. En las líneas 12 a 17 se llevan a cabo una serie de inicializaciones, incluyendo la de `ARMCI` (`ARMCI_Init()` en la línea 12) y algunas variables. Al finalizar este bloque de inicializaciones se realiza una sincronización (`MP_BARRIER()`, línea 17) para asegurar que todas los procesos estén listos. Luego se ejecuta el código principal en paralelo, finalizando con otra sincronización que asegura que todos lo procesos han obtenido los resultados (`MP_BARRIER()`, línea 21).

Al final del código se libera la memoria utilizada (`ARMCI_Free()`, línea 24) y se finalizan los entornos de ejecución, tanto de `ARMCI` (`ARMCI_Finalize()`, línea 25), como de la librería usada (`MP_Finalize()`, línea 26). Como se puede comprobar, este código es independiente de la librería de paso de mensajes que se haya elegido para ser usada con `ARMCI`. Como la elección de esta librería se realiza en tiempo de compilación y puede indicarse directamente desde línea de comandos, no es necesario modificar la aplicación `ARMCI` creada para usar una u otra librería.

```
1  /* ARMCI is message-passing ambivalent -
2     we define macros for common MP calls*/
3  #ifdef PVM
4  #   include <pvm3.h>
5  #   ifdef CRAY
6  #     define MPGROUP          (char *)NULL
7  #     define MP_INIT(arc,argv)
8  #   else
9  #     define MPGROUP            "mp_working_group"
10 #     define MP_INIT(arc,argv) pvm_init(arc, argv)
11 #   endif
12 #   define MP_FINALIZE()    pvm_exit()
13 #   define MP_BARRIER()     pvm_barrier(MPGROUP,-1)
14 #   define MP_MYID(pid)     *(pid) = pvm_getinst(MPGROUP,
       pvm_mytid())
15 #   define MP_PROCS(pproc) *(pproc)=(int)pvm_gsize(MPGROUP)
16 #elif defined(TCG)
17 #   include <sndrcv.h>
18     long tcg_tag =30000;
19 #   define MP_BARRIER()       SYNCH_(&tcg_tag)
20 #   define MP_INIT(arc,argv) PBEGIN_((argc),(argv))
21 #   define MP_FINALIZE()      PEND_()
22 #   define MP_MYID(pid)       *(pid)    = (int)NODEID_()
23 #   define MP_PROCS(pproc)    *(pproc) = (int)NNODES_()
24 #else
25 #   include <mpi.h>
26 #   define MP_BARRIER()       MPI_Barrier(MPLCOMM_WORLD)
27 #   define MP_FINALIZE()      MPI_Finalize()
28 #   define MP_INIT(arc,argv) MPI_Init(&(argc),&(argv))
29 #   define MP_MYID(pid)       MPI_Comm_rank(MPLCOMM_WORLD,
       (pid))
30 #   define MP_PROCS(pproc)    MPI_Comm_size(MPLCOMM_WORLD,
       (pproc));
31 #endif
```

Listado 1.6: Ejemplo de definiciones `ARMCI`

```
1  main(int argc, char *argv[]) {
2    int nproc, me = 0;
3    /* ARMCI */
4    void **ptr;
5    double **ptr_loc;

7    MP_INIT(arc,argv);
8    MP_PROCS(&nproc);
9    MP_MYID(&me);

11   /* Initialize ARMCI and initialization code */
12   ARMCI_Init();
13   ptr = (void **)malloc(nproc * sizeof(void *));
14   ARMCI_Malloc(ptr, proc_bytes);
15   ...
16   /* barrier to ensure all initialization is done */
17   MP_BARRIER();

19   /* main code */
20   ...
21   MP_BARRIER();

23  /* done */
24   ARMCI_Free(ptr[me]);
25   ARMCI_Finalize();
26   MP_FINALIZE();
27 }
```

Listado 1.7: Ejemplo de una aplicación típica `ARMCI`

1.3.5. GASNet

`GASNet` (**G**lobal **A**ddress **S**pace **NET**working) [18] es una capa de red de bajo nivel independiente del lenguaje que ofrece una serie de primitivas de comunicación de alto rendimiento independientes de la red.

Sus principales objetivos son el alto rendimiento, la portabilidad de la interfaz y la expresividad. `GASNet` está pensada para la implementación de un lenguaje paralelo SMPD (**S**ingle **P**rogram, **M**ultiple **D**ata) de espacio global de direcciones, tales como `UPC`, [31], `Titanium` [156] y `Co-Array Fortran` [61].

`GASNet` se divide en dos capas para maximizar la portabilidad sin sacrificar rendimiento. El nivel inferior es una interfaz bastante general denominada `GASNet Core API`, que se sitúa directamente sobre el nivel físico de la red de comunicaciones. Esta capa contiene las primitivas básicas, que se caracterizan por ser lo más reducidas y generales posible. Esta API se implementada directamente sobre cada plataforma y se basa en gran medida en el paradigma de *mensajes activos* [106] [105].

Por encima del `GASNet core API` existe una interfaz más expresiva llamada `GASNet extended API`, que da soporte a operaciones de alto nivel, tales como acceso remoto a memoria y varias operaciones colectivas. A partir de esta capa se asientan, sucesivamente, el sistema de ejecución específico del compilador y la generación de código.

Generalmente se usan llamadas a funciones de la API extendida para implementar la mayoría de sus comunicaciones, asegurando de este modo un rendimiento óptimo a través de diferentes plataformas. Sin embargo, también se permite el uso de la interfaz de mensajes activos del núcleo para implementar operaciones de comunicación no triviales, específicas del lenguaje o del compilador, que no serían apropiadas mediante el uso de una API independiente del lenguaje (tales como implementar bloqueos distribuidos a nivel de lenguaje, recolectores de basura distribuidos, asignación de memoria distribuida, etc.). La característica de mensajes activos del núcleo ofrece un potente mecanismo de extensión que permite implementar una amplia variedad de operaciones de comunicación especializada.

`GASNet`, gracias a su independencia del medio físico, da soporte a una amplia variedad de arquitecturas paralelas y sistemas operativos. Como ejemplos se pueden citar equipos SMP (Origin 2000, multiprocesadores Linux/Solaris , etc.), clusters de uniprocesadores (clusters linux con myrinet, infiniband, via, etc.) y clusters SMP (IBM SP-2, Compaq Alphaserver,

Linux CLUMPS, etc.). Además, GASNet ofrece facilidad al trabajar en nuevos medios físicos, permitiendo implementaciones rápidas, flexibilidad en el paradigma de mensajes (sondeos, interrupciones, híbridos, ...), etc.

Por otro lado, la independencia de GASNet ofrece compatibilidad con varios lenguajes y compiladores de espacio global de direcciones, tales como los citados UPC, Titanium y Co-Array Fortran. Estas características refuerzan uno de los objetivos principales de GASNet: obtener simultáneamente portabilidad y rendimiento.

El listado 1.8 muestra un fragmento de un código GASNet que mide el rendimiento de los mensajes activos. El código comienza con una llamada a la macro GASNET_BEGIN_FUNCTION() (línea 2). Esta macro se puede colocar opcionalmente al principio de las funciones que realicen llamadas repetitivas a GASNet, para amortizar en algunas implementaciones el sobrecosto de descubrir los threads. Para la medida del rendimiento, el código realiza dos pasadas, una de calentamiento entre las líneas 4 y 15 y la pasada efectiva entre las líneas 16 a 24, en la que lleva a cabo la toma de tiempos para medir el rendimiento (para mayor claridad, hemos eliminado del código las rutinas de toma de tiempos).

Los mensajes activos se gestionan a través de gasnet_AMRequestShortM() (líneas 7, 11 y 19), que envía una petición corta de mensaje activo al nodo destino, pasándole M argumentos (en este caso, M es 0). Estas llamadas se encierran dentro de la macro GASNET_Safe(), que comprueba que las funciones se hayan ejecutado de forma correcta (en ese caso devuelven GASNET_OK) y, de no ser así, gestionan el error y finalizan la ejecución.

Tras las peticiones de mensajes activos, se realizan llamadas a la macro GASNET_BLOCKUNTIL() (líneas 10, 13 y 21). Esta macro implementa, de la forma más eficiente para el núcleo actual de GASNet, un bucle bloqueante de espera que se ejecuta hasta que se cumpla la condición indicada. El bloqueo no sólo se lleva a cabo en la ejecución del thread actual, sino también en los servicios de la red. Al finalizar cada bloque se lleva a cabo una sincronización mediante la macro BARRIER() (líneas 15 y 24), que encapsula llamadas a las funciones gasnet_barrier_notify() (notifica la barrera) y a gasnet_barrier_wait() (espera a que ésta se complete), todo esto con la gestión de errores anteriormente comentada mediante el uso de la macro GASNET_Safe().

```
1   void doAMShort() {
2     GASNET_BEGIN_FUNCTION();

4     if (sender) { /* warm-up */
5       flag = 0;
6       for (i=0; i < iters; i++) {
7         GASNET_Safe(gasnet_AMRequestShort0(peer,
8                     hidx_ping_shorthandler_flood));
9       }
10      GASNET_BLOCKUNTIL(flag == iters);
11      GASNET_Safe(gasnet_AMRequestShort0(peer,
12                  hidx_ping_shorthandler));
13      GASNET_BLOCKUNTIL(flag == iters+1);
14    }
15    BARRIER();
16    if (sender) {
17      flag = -1;
18      for (i=0; i < iters; i++) {
19        GASNET_Safe(gasnet_AMRequestShort0(peer,
20                    hidx_ping_shorthandler));
21        GASNET_BLOCKUNTIL(flag == i);
22      }
23    }
24    BARRIER();
25  }
```

Listado 1.8: Ejemplo de una aplicación implementada con `GASNet`

1.4. Lenguajes PGAS

El modelo de programación *PGAS* (*P*artitioned *G*lobal *A*ddress *S*pace) [155] es un paradigma SPMD que proporciona facilidad de uso a través de un espacio compartido de direcciones globales, mientras que presta una especial atención al rendimiento a través de la localidad.

Los lenguajes PGAS combinan lo mejor de la programación en memoria compartida y paso de mensajes. En estos lenguajes, el espacio global de memoria está lógicamente dividido de modo que una porción es local a cada procesador. Es esta característica la que supone una novedad del modelo PGAS, ya que porciones del espacio de memoria compartido pueden tener afinidad con un determinado thread, por lo que es posible explotar la localidad de los datos.

Los lenguajes PGAS típicamente son utilizados en máquinas de memoria distribuida, implementando el espacio virtual de direcciones mediante librerías de comunicaciones unidireccionales tales como `ARMCI` o `GASNet`. Desde hace algunos años, el modelo PGAS ha ido ganando importancia hasta que, en la actualidad, lenguajes basados en este modelo son omnipresentes. El modelo PGAS es la base de lenguajes como `UPC` [31], `Co-array Fortran` [117] y `Titanium` [156].

1.4.1. Co-Array Fortran

`Co-Array Fortran` (`CAF`) [117] es un conjunto de extensiones para `Fortran 95` que convierten a éste en un lenguaje SPMD paralelo eficiente y robusto. Estas extensiones poseen una notación simple y explícita, con una sintaxis independiente de la arquitectura y similar a la de `Fortran`, teniendo como fin el de poder expresar la descomposición de los datos, como normalmente ocurre en los lenguajes de paso de mensajes.

Los programas escritos en `Co-Array Fortran` son muy similares a los programas `Fortran` secuenciales y sólo se requiere que los usuarios aprendan unas pocas reglas nuevas para poder llevar a cabo la distribución de trabajo y de datos. El éxito que este lenguaje ha tenido ha sido tal que será incluido en el próximo estándar de `Fortran` [118].

`Co-Array Fortran` permite poder realizar referencia a múltiples instancias cooperantes de un programa SPMD (lo que se denomina *imagen*) a través de un nuevo tipo de dimensión de vector denominada *co-array*. Al declarar un variable con una dimensión *co-array*, el usuario especifica

que cada imagen del programa contendrá una copia de la variable. Cada imagen podrá entonces acceder a instancias remotas de la variable indexando dentro de la dimensión *co-array* índices que hacen referencia al espacio lógico de imágenes. La coordinación entre imágenes cooperantes se lleva a cabo mediante las rutinas de sincronización provistas por este lenguaje.

La gran ventaja de `Co-Array Fortran` es la simplificación de comunicaciones, ofreciendo los *co-array* una elegante abstracción para las transferencias de datos, comparados con las comunicaciones uni o bidireccionales. Sin embargo, a pesar de que provee una vista más coherente de los datos distribuidos, comparado con librerías de paso de mensajes, todavía se requiere la intervención del usuario para tratar la fragmentación de los vectores en diferentes trozos para cada procesador, llegando en algunos aspectos al mismo nivel de detalle que, por ejemplo, se tendría con `MPI`.

El listado 1.9 recoge un ejemplo de un programa que calcula el número π de forma iterativa, programado con `Co-Array Fortran`. En las líneas 5 y 6 podemos observar el uso de la extensión *co-array* mediante la que el programador puede expresar la distribución especificando la relación entre las imágenes de memoria. En este código también se localizan llamadas a la función `sync_all` (líneas 17, 26 y 33), una barrera global que requiere que todas las operaciones que se encuentran antes de la llamada en todas las imágenes tengan que ser completadas antes que cualquier imagen avance más allá de la llamada.

1.4.2. UPC

Unified Parallel C (`UPC`) [151, 31, 86] es una extensión explícitamente paralela de `ANSI C` a un estilo de computación PGAS [63, 62]. `UPC` toma como base los conceptos y características de `C` y añade paralelismo, con un acceso global a memoria, diferenciando lo que es local y remoto y permitiendo el acceso a memoria remota para leer y escribir, usando para ello sentencias simples. El lenguaje aporta un modelo uniforme de programación, tanto para máquinas de memoria compartida como distribuida.

Al programador la memoria se le presenta como único espacio de memoria compartido y particionado, donde las variables pueden ser leídas y escritas directamente por cada procesador, si bien cada variable está físicamente asociada con un único procesador. `UPC` utiliza el modelo de computación SPMD en el que la cantidad de paralelismo está establecida en el momento que el programa empieza a ejecutarse, normalmente con un único thread de

```
1  program compute_pi
2  implicit none
3  double precision :: pi,psum,x,w
4  integer          :: me,nimg,i
5  double precision, save :: mypi[*]
6  integer,          save :: n[*]

8  nimg = num_images()
9  me   = this_image()

11 if (me==1) then
12    write(6,*) 'Enter number of intervals'
13    read( 5,*) n[@]
14    write(6,*) 'number of intervals = ',n[@]
15    n[:] = n[@]
16 endif
17 call sync_all(1)

19 w = 1.d0/n[@]
20 psum = 0.d0
21 do i= me,n[@],nimg
22    x = w * (i - 0.5d0)
23    psum = psum + 4.d0/(1.d0+x*x)
24 enddo
25 mypi[@] = w * psum
26 call sync_all()

28 if (me==1) then
29    pi = sum(mypi[:])
30    pi = sum(mypi[1:nimg])
31    write(6,*) 'computed pi = ',pi
32 endif
33 call sync_all(1)
34 end
```

Listado 1.9: Ejemplo del cálculo de π mediante un código `Co-Array Fortran`

ejecución por procesador.

El uso de la palabra reservada `shared` produce que los elementos linealmente ordenados de un vector sean distribuidos entre las diferentes instancias del programa (o threads) de forma cíclica o cíclica por bloques. Este mecanismo ofrece una visión más global de los vectores de los usuarios. En este sentido, `UPC` también brinda una ligera mejora de la visión del control al introducir una nueva estructura de bucle, el *bucle* ***`upc_forall`***, donde cada iteración global de un bucle `for` de estilo `C` es asignada a los threads siguiendo una *expresión de afinidad*.

Sin embargo, puede que la característica más útil de `UPC` sea el soporte de punteros dentro del espacio global compartido de direcciones. Los punteros son declarados de forma `private` (local a un thread) o bien `shared`, y pueden apuntar a datos que también pueden ser privados o compartidos. Esto conlleva una importante abstracción que favorece la programación del espacio de direcciones compartido.

El listado 1.10 recoge otra versión del algoritmo de cálculo del número π, esta vez escrita en `UPC`. En este ejemplo, los threads llaman colectivamente a la función `upc_all_lock_alloc()` (línea 13) para crear el bloqueo `l`. La tarea de calcular π se divide entre los threads usando el bucle `upc_forall` (línea 15). Cada thread acumula su porción de la suma total en su propia variable privada `local_pi`, para luego, ser sumada y obtener así el resultado global en la línea 20. Para evitar que las actualizaciones de la variable se superpongan, todos los threads utilizan un bloqueo mediante la función `upc_lock` (línea 19), que luego libera con la función `upc_unlock` (línea 21). Tras esto, se ejecuta un barrera para que los distintos threads esperen por aquellos que aún están actualizando la variable. Para finalizar, un sólo thread realiza la llamada para liberar el bloqueo (`upc_lock_free`, línea 25).

1.4.3. Titanium

`Titanium` [156, 82] es un dialecto explícitamente paralelo de `Java`, desarrollado en Berkeley con el objetivo de dar soporte a aplicaciones científicas de altas prestaciones en sistemas multiprocesador, entre los que se incluyen supercomputadores masivamente paralelos y clusters de memoria distribuida, con uno o varios procesadores por nodo. Algunas de sus características son, la seguridad, portabilidad y soporte para construir estructuras de datos complejas.

`Titanium` añade un gran conjunto de características a `Java` con el fin de

```
1  //Numerical Integration
2  #include <upc_relaxed.h>
3  #include <math.h>

5  #define N 1000000
6  #define f(x) (1.0/(1.0+x*x))

8  upc_lock_t *l;
9  shared float pi = 0.0;
10 void main(void) {
11   float local_pi=0.0;
12   int i;
13   l=upc_all_lock_alloc();

15   upc_forall(i=0;i<N;i++; i)
16     local_pi +=(float) f((.5+i)/(N));
17   local_pi *= (float) (4.0 / N);

19   upc_lock(l);
20   pi += local_pi;
21   upc_unlock(l);

23   upc_barrier; // Ensure all is done
24   if(MYTHREAD==0) printf("PI=%f\n",pi);
25   if(MYTHREAD==0) upc_lock_free(l);
26 }
```

Listado 1.10: Ejemplo del cálculo de π mediante un código UPC

facilitar la programación al usuario final. Entre ellas se puede destacar:

- un modelo de control paralelo explícito SPMD
- vectores multidimensionales flexibles y eficientes, que soportan iteradores, subvectores y métodos de copia
- sobrecarga de operadores
- optimización agresiva gracias a la ejecución sin orden de las iteraciones de bucles
- prevención durante la compilación de interbloqueos en la sincronización de barreras
- librería de rutinas útiles de operaciones colectivas y de sincronización
- plantillas (clases parametrizadas)
- Clases inmutables definidas por el usuario, también conocidas como clases ligeras ("lightweigth classes") o "value classes"
- tipos internos para representar puntos en varias dimensiones, rectángulos y dominios generales
- gestión de memoria basada en zonas (además del recolector de basura de `Java`)
- sistema de tipos para expresar o inferir localidad y atributos compartidos de estructuras de datos distribuidas
- etc.

`Titanium` proporciona abstracción al ofrecer un espacio de memoria global. En este espacio toda la memoria es afín a un procesador y puede ser controlable por el usuario. No obstante, los procesos paralelos pueden referenciar directamente la memoria de otros procesadores para leer, escribir valores o realizar transferencia de datos. La portabilidad de los programas de `Titanium` permite que puedan ejecutarse sin modificaciones en sistemas uniprocesadores, de memoria compartida y distribuida, si bien puede ser necesario en estos últimos algunos ajustes para las estructuras de datos.

`Titanium` dispone de la mayoría de características que los desarrolladores de programas secuenciales podrían esperar. El hecho de que sea un superconjunto de `Java` y de ser un lenguaje orientado a objetos aumenta

sus capacidades para separar los algoritmos de su implementación y disponer de mecanismos para crear abstracciones de datos. Sin embargo, como otros lenguajes PGAS, cuenta con la desventaja de dar soporte a un modelo SPMD que dificulta una visión global de memoria de las estructuras de datos y control, así como la expresión de paralelismo general.

El listado 1.11 presenta un ejemplo de un código realizado en `Titanium`. El objetivo del mismo es el cálculo de parcelas adyacentes, dentro del algoritmo paralelo adaptivo de refinamiento de mallas 3D [128]. Hemos elegido la porción de código que se muestra porque, en ella, se reflejan algunas de las características más útiles de `Titanium`, como es la habilidad de manipular directamente *puntos* (`Points`) que consisten en una t-upla de enteros), *dominios* (`Domains`) o conjunto de puntos y lo que en `Titanium` han denominado `RectDomain`, conjunto de puntos que pueden ser representados por un entero de inicio, fin y desplazamiento o salto en cada dimensión. Por ejemplo, en cálculos sobre bordes de regiones, es útil calcular un conjunto de celdas sobre las que se debe iterar, lo cual puede ser fácilmente representado en `Titanium` como `Domains` o `RectDomain`.

El código expuesto está formado por dos bucles. En el primero de ellos, el bucle exterior de la línea 1, itera sobre un vector de datos preprocesados. El bucle interior de la línea 3 itera sobre cada parcela del nivel l que "pertenece" al proceso p. La adyacencia se determina por fuerza bruta, comparando los dominios.

1.4.4. Global Arrays

`Global Arrays` [115, 72] ofrece un estilo de programación similar al de memoria compartida para programación en máquinas de memoria distribuida de forma portable y eficiente. Las operaciones básicas de memoria compartida a las que `Global Arrays` da soporte son `get`, `put`, `scatter` y `gather`, así como las operaciones atómicas `read-and-increment`, `accumulate` (operación de reducción que combina datos locales con otros que se localizan en memoria compartida) y operaciones de bloqueo.

Las operaciones de transferencia de datos usan una interfaz basada en índices de vector, en vez de direccionar los datos compartidos. De este modo, se libera al usuario de tener que especificar los procesos donde residen los datos compartidos referenciados, simplemente se ofrece una visión global de las estructuras de datos. Este mayor nivel de abstracción de la API hace más fácil el uso de `Global Arrays`, sin comprometer el control de la localidad de

```
1  foreach (pwithinAMR.processes.domain()){
2    AMRProcessap = AMR.processes[p];
3    foreach(iiwithinap.levels[l].patches.domain()) {
4      Patch otherPatch = ap.levels[l].patches[ii];
5      if (this != otherPatch){
6        RectDomain<3> b = gdomain * otherPatch.domain;
7        if (!b.isNull()) {
8          RelatedPatchrp = newRelatedPatch();
9          rp.domain = b;
10         rp.patch = otherPatch;
11         adjacentList.push(rp);
12       }
13     }
14   }
15 }
16 adjacentPatches = adjacentList.toArray();
```

Listado 1.11: Ejemplo de un código `Titanium`

los datos. Es la librería quien internamente realiza la traducción del índice global del vector a una dirección y entonces transfiere los datos entre los procesos correctos. Si lo necesita, el usuario siempre puede determinar dónde está localizado un elemento o sector de un vector y qué proceso o procesos son los propietarios de los datos de las secciones especificadas del vector.

La librería de `Global Arrays` soporta dos estilos de programación: paralelismo de tareas y de datos. El primero de ellos presenta un modelo de computación basado en la copia explícita de memoria remota, es decir, la porción de memoria remota compartida debe ser copiada dentro del área de memoria local antes de que pueda ser usada por el correspondiente proceso. En cuanto al modelo de paralelismo de datos, se realiza mediante un conjunto de funciones colectivas que operan sobre los vectores globales o partes de ellos. Si alguna operación de comunicación entre procesadores debe ser efectuada, la librería realiza copias de memoria remota o, en un número menor de casos, operaciones de paso de mensajes.

`Global Arrays` ha sido diseñado para complementar más que para sustituir al modelo de paso de mensajes. El programador es libre de usar en el mismo programa ambos paradigmas, tanto el de memoria compartida como el de paso de mensajes, obteniendo así la ventaja de librerías de paso de mensajes existentes, como `MPI`, con quien es compatible. De este modo, `Global Arrays` hace uso de la visión global/compartida más conveniente

para los vectores multidimensionales, pero puede usar también el modelo de `MPI` allí donde sea necesario.

Otro punto fuerte de `Global Arrays` es el control explícito de la localidad de datos y la granularidad, mediante el modelo *get-compute-put*, al contrario que otros modelos, que introducen sobrecargas con comunicaciones no transparentes. Por otro lado, la aproximación basada en librerías permite que no se tenga que confiar en las optimizaciones realizadas por el compilador para obtener un buen rendimiento. Sin embargo, la mayor desventaja y limitación de `Global Arrays` es que sólo es posible usarlo en estructuras de datos de tipo vector.

El listado 1.12 muestra un ejemplo de un código `Global Array` que implementa un algoritmo de multiplicación de matrices. Debido al elevado número de líneas que componen el código original, en este ejemplo principalmente se presentan sólo las llamadas a las funciones de `Global Array`. Entre ellas, destacan las funciones que devuelven el identificador y el número de elementos que componen el grupo (`GA_Nodeid()` y `GA_Nnodes()`, respectivamente, en la línea 7).

A partir de esto se configura el reparto en las matrices (líneas 10 a 14) y, con esto datos, se crea el vector global `g_a` mediante la función `NGA_Create()` en la línea 17, para luego ser duplicado mediante la función `GA_Duplicate()` en las líneas 18 y 19. Los accesos a estos vectores se realizan mediante las funciones `NGA_Put()` y `NGA_Get()`, sincronizando cuando es necesario mediante `GA_Sync()`, como en la línea 28. Al finalizar, la memoria de los vectores se libera con la función `GA_Destroy()`, en las líneas 43 a 45.

1.5. eSkel

`eSkel` (**e**dinburgh **Ske**leton **l**ibrary) [11, 36] es una librería de programación paralela desarrollada en la Universidad de Edimburgo. Esta librería no alcanza la relevancia e impacto de otros lenguajes y herramientas paralelas estudiados en este capítulo, sin embargo, hemos considerado conveniente incluir una descripción de `eSkel` debido a que su estudio ha estado presente en nuestro trabajo, y algunas de sus características han servido de base en el diseño de nuestro lenguaje.

Por otro lado, durante la realización de esta tesis se llevó a cabo una estancia en el Centro de Supercomputación de Edimburgo (EPCC) [64], bajo la dirección de Murray Cole, creador de `eSkel` y una de las personalidades

```
int dims[NDIM], chunk[NDIM], ld[NDIM];
int lo[NDIM], hi[NDIM], lo1[NDIM], hi1[NDIM];
int lo2[NDIM], hi2[NDIM], lo3[NDIM], hi3[NDIM];
int g_a, g_b, g_c, i, j, k, l;

/* Find local processor ID and the number of processors */
int me=GA_Nodeid(), nprocs=GA_Nnodes();

/* Config. array dims. Force an unequal data distrib. */
for(i=0; i<NDIM; i++) {
   dims[i]  = TOTALELEMS;
   ld[i]= dims[i];
   chunk[i] = TOTALELEMS/nprocs-1;
}

/* create a global array g_a and duplicate it (g_b, g_c) */
g_a = NGA_Create(C_DBL, NDIM, dims, "array A", chunk);
g_b = GA_Duplicate(g_a, "array B");
g_c = GA_Duplicate(g_a, "array C");

 /* initialize data in matrices a and b copy data*/
...
if (me==0) {
   NGA_Put(g_a, lo1, hi1, a, ld);
   NGA_Put(g_b, lo1, hi1, b, ld);
}
/*  Synch. all processors to make sure everyone has data */
GA_Sync();

/* Determine which block of data is locally owned */
NGA_Distribution(g_c, me, lo, hi);
/* Get the blocks from g_a and g_b needed to compute this
block in g_c and copy them into the local buffers a, b. */
...
NGA_Get(g_a, lo2, hi2, a, ld);
NGA_Get(g_b, lo3, hi3, b, ld);
...
/* Copy c back to g_c */
NGA_Put(g_c, lo, hi, c, ld);
verify(g_a, g_b, g_c, lo1, hi1, ld);

/* Deallocate arrays */
GA_Destroy(g_a);
GA_Destroy(g_b);
GA_Destroy(g_c);
```

Listado 1.12: Ejemplo de un código `Global Array`

más relevantes en el campo del uso de esqueletos en la programación paralela [34, 35].

La librería `eSkel` proporciona un conjunto de esqueletos en `MPI` y `C`, aportando un nivel de abstracción en operaciones colectivas a los programadores de `MPI`. De esta forma, ofrece una interfaz de funciones de alto nivel que implementan las operaciones colectivas más habituales.

El acceso a los datos sigue la sintaxis y semántica convencional de `C/MPI`. Un concepto clave en `eSkel` es su *modelo de datos* (`eDM`: **e**Skel **D**ata **M**odel), definiéndose los conceptos *`eDM` átomo*, unidad de transferencia entre la función del esqueleto y los procesos que componen cada actividad y *`eDM` colección*, unidad de transferencia de datos entre un programa que invoca y la función del esqueleto que es invocada. Ambos casos pueden producirse a nivel local o global.

Los procesadores se agrupan y reagrupan siguiendo la semántica del esqueleto. `eSkel` proporciona una serie de funciones que permiten obtener los datos de cada grupo (referencia al comunicador, tamaño del grupo, rango, etc.). Cada grupo de procesadores creado de esta forma se denomina actividad, permitiéndose el anidamiento mediante una pila variable de actividades.

Las actividades interactúan con el sistema e, indirectamente, con otras actividades, mediante el uso de las funciones:

- `Give` (la actividad transfiere un `eDM` al esqueleto actual)
- `Take` (se recibe un átomo `eDM` desde el esqueleto actual a la actividad llamante)
- `Exchange` (permite el intercambio de datos entre procesos asociados, encapsulando las llamadas `Give` y `Take` y codificándolas en la secuencia correcta para evitar interbloqueos).

Además de estas funciones, el tipo de datos `eSkel_atom_t` permite un mecanismo genérico por el cual las funciones de las actividades obtienen una interfaz con los resultados de las comunicaciones implícitas.

Las interacciones orientadas a datos ocurren al entrar o salir de las llamadas a funciones de `eSkel` y en la interacción de las llamadas `Give`, `Take` y `Exchange`. Algunos esqueletos realizan llamadas implícitas a estas funciones, sin embargo, en otros casos se requiere que las actividades realicen llamadas explícitas a las mismas. La sintaxis de estas funciones es genérica, pero cada esqueleto define su propia semántica.

Los esqueletos que eSkel proporciona se centran en dar soporte a paralelismo de granjas, *pipelines* y de tipo *butterfly*. Para cada uno de estas situaciones eSkel dispone de varias funciones que implementan el caso general, añadiendo también funciones que optimizan algunas situaciones específicas. eSkel no tiene una política por defecto de balanceo de carga y cada esqueleto requiere que se le especifique su política.

El listado 1.13 muestra una porción de un código eSkel que hemos desarrollado. En este ejemplo, mediante un esqueleto de granja, se implementa el cálculo de los números primos comprendidos entre 2 y un valor N. En la línea 1 se encuentra la definición de la función que ejecuta cada tarea, usando para ello el tipo átomo de eSkel (eSkel_atom_t), mientras que en la línea 26 se localiza la llamada a la función del esqueleto, SimpleFarm1for1. Este esqueleto es un constructo de granja de tareas con equilibrado de cargas incluido, que realiza una distribución de forma que cada átomo de tarea de la entrada produzca exactamente un átomo resultante.

1.6. HPF

Más de 40 organizaciones forman el Foro de HPF (**H**igh **P**erformance **F**ortran) [149, 94]. Este foro definió una serie de extensiones que denominaron HPF, sobre el estándar ISO de Fortran para tratar problemas resultantes de escribir datos en programas paralelos en aquellas arquitecturas donde la distribución de datos influye en el rendimiento.

En concreto, estas extensiones buscan los siguientes objetivos:

- dar soporte al paralelismo de datos
- portabilidad entre arquitecturas
- alcanzar un alto rendimiento en máquinas con un coste no uniforme de acceso a memoria (sin perder rendimiento en el resto de máquinas)
- interfaz e interoperabilidad abierta con otros lenguajes y otros paradigmas paralelos
- etc.

Al igual que en otros casos, HPF busca un alto grado de portabilidad, no sólo la posibilidad de compilar un código en diferentes máquinas, sino también en cuanto a la eficiencia se refiere.

```
1  eSkel_atom_t *work (eSkel_atom_t *task) {
2    int num, sol,value;

4    value = ((int *) (task->data))[0];
5    if (value == ERROR) {
6      ((int *) (task->data))[0] = ERROR;
7      tasksdone++;
8      return task;
9    }

11   sol = OK;
12   for (num = 2; num <= ((value/2)+1); num++) {
13     if ((value %num) == 0) {
14       sol = ERROR;
15       break;
16     }
17   }
18   ((int *) (task->data))[0] = sol;
19   tasksdone++;

21   return task;
22 }

24 ...

26 SimpleFarm1for1 (work, (void *) (tasks+myrank()*num_task),
27                  1, num_task, SPLOCAL, MPI_INT,
28                  (void *) ulam, 1, &outmul, SPLOCAL,
29                  MPI_INT, num_task, mycomm());
```

Listado 1.13: Ejemplo de un código eSkel

Las extensiones de HPF dan soporte a una visión global de vectores distribuidos y ofrece un único hilo lógico de ejecución. Sus directivas permiten al usuario orientar la distribución y alineamiento de vectores, la organización de bucles y otros detalles relevantes en computación paralela. El compilador de HPF implementa el código del usuario generando un programa SPMD en el que el código generado por el compilador y su motor de ejecución gestionan los detalles de la implementación de los vectores distribuidos y la comunicación entre procesadores.

Sin embargo, HPF adolece de algunas deficiencias. Por ejemplo, su modelo de ejecución SPMD y un único hilo de ejecución resultan adecuados para expresar paralelismo de un nivel, pero no para expresar paralelismo de tareas o paralelismo anidado. HPF también experimenta otras carencias, como una falta de transparencia de su modelo de ejecución, siendo difícil tanto para el usuario como para el compilador determinar cómo un código debería ser implementado. Todo esto y otros factores han ocasionado que HPF no haya tenido el éxito esperado [92].

El ejemplo del listado 1.14 recoge una porción de una aplicación HPF que muestra algunas de las características de este lenguaje. En la primera línea, la directiva GEOMETRY permite al usuario especificar de forma genérica un mapeado y hacer uso del mismo par aplicarlo a varios vectores. Este ejemplo tiene un único bucle marcado con la directiva INDEPENDENT (bucle exterior de la línea 8). Dentro de este bucle, el valor privado de JCELL0 es asignado a cada procesador, asegurando siempre que se trata de un cómputo local. En las líneas 11 y 13 se sitúan dos bucles anidados, ambos privados, siendo la variable JCELL procesada localmente en cada procesador, por lo que se minimizan las comunicaciones.

1.7. ZPL

ZPL [25, 141] es un lenguaje paralelo orientado al procesamiento de vectores, diseñado en la Universidad de Washington con el fin de obtener una rápida ejecución, tanto en sistemas secuenciales como paralelos. Este lenguaje se propone como una alternativa para la programación de supercomputadores y grandes clusters con una eficiencia similar a la que tendrían los correspondientes programas *ad-hoc* basados en paso de mensajes.

En cuanto a la portabilidad, ZPL compila sobre ANSI C con llamadas a la librería de comunicación que elija el usuario (por ejemplo, MPI). De este modo, ZPL consigue una alta portabilidad. Esta portabilidad también

```
1  !HPF$ GEOMETRY G(*, CYCLIC)
2        REAL FX(100,100), FY(100,100), FZ(100,100)
3  !HPF$ DISTRIBUTE (G) :: FX,FY,FZ
4        REAL FXP(100,16,100), FYP(100,16,100)
5  !HPF$ DISTRIBUTE FXP(*,*, BLOCK) FYP(*,*, BLOCK)
6        INTEGER CELL, ATOM, MAP(1000), NACELL(1000)

8  !HPF$ INDEPENDENT (CELL) ON FX(1,CELL)
9        DO CELL=1,100
10         JCELL0 = 16*(CELL-1)
11         DO NABOR = 1, 13
12           JCELL = MAP(JCELL0+NABOR)
13           DO ATOM=1, NACELL(CELL)
14             FX(ATOM, CELL) = FX(ATOM, CELL) + FXP(ATOM,
       NABOR, JCELL)
15             FY(ATOM, CELL) = FY(ATOM, CELL) + FYP(ATOM,
       NABOR, JCELL)
16           ENDDO
17         ENDDO
18       ENDDO
```

Listado 1.14: Ejemplo de un código HPF

se entiende en el sentido de rendimiento, manteniendo las aplicaciones ZPL un rendimiento similar en diversas plataformas, mejorando en este sentido a algunas aplicaciones codificadas directamente en MPI.

ZPL ofrece una característica única dentro de los lenguajes de visión global, en el sentido que soporta un modelo de ejecución transparente, utilizando el concepto conocido como *WYSIUWYG* (What You See Is What You Get) [26], en cuanto su sintaxis identifica inherentemente aquellas operaciones que inducen comunicaciones. Esta característica visual simplifica la evaluación en primer orden del coste del programa paralelo.

Las operaciones tradicionales están semánticamente restringidas en ZPL para ser aplicables únicamente a aquellas expresiones de vectores que siguen una determinada alineación. Si un vector no sigue una de estas alineaciones, entonces se debe aplicar una serie de operadores de vectores que sirven para expresar diferentes patrones de accesos.

ZPL ofrece paralelismo con una visión global de memoria en vectores distribuidos. Sin embargo, su modelo de ejecución sólo soporta programas con un único nivel de paralelismo de datos en un momento dado, lo cual limita su generalidad.

El listado 1.15 recoge un ejemplo de un código ZPL que implementa una versión del algoritmo iterativo de Jacobi [81]. En esta versión se calcula la iteración de Jacobi en 4 puntos en una matriz A de dimensión $n \times n$, donde cada elemento es reemplazado por la media de sus cuatro vecinos. Este ejemplo no muestra las características más avanzadas de ZPL [27], pero sirve para ilustrar las propiedades básicas del lenguaje, como conceptos de alto nivel tales como soporte para la manipulación de vectores a través de regiones, operaciones sobre vectores, expresión implícita de paralelismo de datos y la gestión por parte del compilador de los detalles de las comunicaciones y sincronizaciones.

Las variables de configuración declaradas al principio del código del listado 1.15 son definidas durante el proceso de carga y permanecen constantes. Estas variables configuran la computación definiendo valores específicos para el programa, como el tamaño del problema. Como el paralelismo es *implícito* en ZPL, en el código no aparecen datos como el número de procesadores. A cada configuración se le asigna un valor por defecto (en el código fuente) que el usuario puede modificar por línea de comandos o mediante un fichero de configuraciones.

En las líneas 8 a 10 se realiza la definición de las regiones que componen el problema. Estas regiones pueden utilizarse de dos formas: en la primera

```
1  program jacobi;
2
3  ---------- Declarations ----------
4  config var
5    n       : integer = 5;      -- problem size
6    epsilon : float   = 0.0001; -- epsilon value/condition
7
8  region
9    R    = [1..n,    1..n   ]; -- problem region
10   BigR = [0..n+1, 0..n+1]; --     with borders
11
12 direction
13   north = [-1,  0]; -- cardinal directions
14   east  = [ 0,  1];
15   south = [ 1,  0];
16   west  = [ 0, -1];
17
18 ---------- Entry Procedure ----------
19 procedure jacobi();
20 var
21   A, Temp : [BigR] float;
22   delta   :        float;
23 [R] begin
24
25   ---------- Initialization ----------
26   A := 0.0;
27   [north of R] A := 0.0;
28   [east  of R] A := 0.0;
29   [west  of R] A := 0.0;
30   [south of R] A := 1.0;
31
32   ---------- Main Computation ----------
33   repeat
34     Temp := (A@north + A@east + A@south + A@west) / 4.0;
35     delta := max<< abs(A-Temp);
36     A := Temp;
37   until delta < epsilon;
38
39 end;
```

Listado 1.15: Ejemplo de un código ZPL

pueden usarse para declarar variables de vectores, como en la línea 21 donde se declaran `A` y `Temp`; por otro lado, también pueden emplearse las regiones para especificar dominios, como en la línea 23. En las líneas 12 a 16 se especifican las coordenadas de la cuadrícula mediante vectores definidos por el usuario, que luego serán utilizadas en conjunción con las regiones mediante el operador `of`, como en la inicialización (líneas 25 a 30), o el operador `@`, durante el acceso a cada punto que rodea a la coordenada actual (línea 34). Las iteraciones se llevan a cabo mediante el bucle `repeat` de la línea 33, cuya región se aplica a cada sentencia del cuerpo del bucle. La línea 35 lleva a cabo el cambio del elemento máximo sobre todos los demás, usando para ello un operador de reducción `<<`.

1.8. Cilk

`Cilk` [69, 142, 33] es un lenguaje paralelo *multithread* basado en `ANSI C` que ha sido diseñado para programación paralela de propósito general, si bien destaca especialmente en paralelismo dinámico altamente asíncrono, que usualmente es difícil de expresar como paralelismo de datos o paso de mensajes. `Cilk` también ofrece una plataforma efectiva para la programación de algoritmos numéricos densos y dispersos. A diferencia de otros sistemas, `Cilk` es algorítmico, en el sentido que el sistema de ejecución emplea un planificador que permite estimar de forma precisa el rendimiento de los programas a partir de medidas abstractas complejas.

Para el programador, los códigos `Cilk` se asemejan a códigos escritos en `ANSI C`, que han sido anotados con operaciones para generar y sincronizar los threads. De hecho, si se eliminan las palabras reservadas de `Cilk`, el resultado es un código `ANSI C` válido con la misma semántica que el código `Cilk`, pero secuencial.

La filosofía de `Cilk` se basa en que el programador debe concentrar sus esfuerzos en estructurar el programa para expresar paralelismo y explotar localidad, dejando que el sistema de ejecución de `Cilk` se encargue de planificar la ejecución de forma eficiente para una plataforma específica. De esta forma, el sistema de ejecución de `Cilk` se ocupa de detalles como el equilibrado de carga, paginación, protocolos de comunicación, etc. De hecho, `Cilk` utiliza técnicas agresivas de compartición y robo de tareas (*work sharing and stealing*) [15, 16] para equilibrado de la carga computacional y para evitar sobrecargar el sistema con demasiado paralelismo.

`Cilk` ha tenido una especial relevancia en el diseño de nuestro compilador.

El compilador de `Cilk`, denominado `cilk2c` [110], implementa un análisis léxico y sintáctico de la gramática de `ANSI C` que se hemos tomado como base. El lenguaje que hemos desarrollado también mantiene algunas semejanzas con el modelo de programación de `Cilk`, que consiste en escribir un código secuencial `ANSI C` y anotarlo con la información necesaria para generar y gestionar el paralelismo. Sin embargo, en nuestro diseño hemos optado por que esta información se introduzca mediante directivas de compilador, en lugar de llamadas a funciones como en `Cilk`, ya que las directivas mantienen la integridad del código secuencial.

```
1  #include <stdlib.h>
2  #include <stdio.h>

4  cilk int fib (int n) {

6    if (n<2) return n;
7    else {
8      int x, y;

10     x = spawn fib (n-1);
11     y = spawn fib (n-2);
12     sync;
13     return (x+y);
14   }
15 }

17 cilk int main (int argc, char *argv[]) {
18   int n, result;

20   n = atoi(argv[1]);
21   result = spawn fib(n);
22   sync;
23   printf ("Result: %d\n", result);
24   return 0;
25 }
```

Listado 1.16: Ejemplo de un código `cilk`

El listado 1.16 presenta un ejemplo de un código `Cilk` que implementa un programa recursivo para calcular el n-ésimo número de la serie de Fibonacci, aunque se trata de un ejemplo puramente didáctico, puesto que existen formas más eficientes de realizar este cálculo [39].

El código resultante es prácticamente similar al secuencial, salvo por el uso de las palabras reservadas de `Cilk`. La palabra reservada `cilk` (líneas 4 y 17) identifica un procedimiento `Cilk`, que es una versión paralela de una función `C`. Dentro de estos procedimientos se usan las palabras reservadas `spawn` (líneas 10, 11 y 21) para generar subprocesos paralelos y `sync` (líneas 12 y 22) para la sincronización tras la finalización. Si suprimimos de este programa las palabras reservadas de `Cilk` mencionadas, el resultado es una aplicación secuencial válida.

1.9. Fortress

`Fortress` [2] es un lenguaje desarrollado por `SUN` dentro del programa HPCS. Su nombre deriva de "*secure Fortran*", debido a que es uno de los objetivos que persigue.

`Fortress` está basado en lenguajes de programación actuales, pero su sintaxis es nueva. Esta sintaxis está algo alejada de los lenguajes de programación tradicionales, e intenta ser lo más parecida a la representación matemática. La idea es que si la notación matemática puede ser entendida por científicos de todo el mundo, el diseño de un lenguaje que admita una sintaxis similar debería facilitar la programación y entendimiento de los códigos por parte de la comunidad científica.

De este modo, `Fortress` permite que se definan ecuaciones, funciones tales como sumas o productos sobre conjuntos, etc., de un modo similar al que se llevaría a cabo en notación matemática. Además, `Fortress` permite el uso de unidades físicas en la definición de variables, pudiendo utilizarse caracteres *unicode* e incorporando las librerías de `Fortress` por defecto un conjunto de dimensiones y unidades.

`Fortress` también soporta operaciones de reducción y definiciones por comprensión, que permiten describir conjuntos de elementos usando reglas que deben cumplir todos sus elementos.

`Fortress` ha sido diseñado como un lenguaje de propósito general, haciendo un especial hincapié en su adecuación a sistemas paralelos. Se pretende que el diseño de códigos paralelos sea simple, tanto para paralelismo de tareas como de datos. Algunos de los constructos paralelos en `Fortress` son implícitos, como los bucles `for`, que son ejecutados en paralelo por defecto.

Las unidades básicas de los códigos `Fortress` son los *objetos* y los

denominados *traits*. Los objetos definen los *campos* y los *métodos*, mientras que los *traits* declaran conjunto de métodos, tanto abstractos como concretos. `Fortress` es un lenguaje interpretado que se ejecuta sobre una `JVM` (***J**ava **V**irtual **M**achine*).

El listado 1.17 recoge el cálculo del número π mediante un código `Fortress`, haciendo uso del algoritmo de la *aguja de Buffon* [137]. Las variables declaradas entre las líneas 7 y 9 (`needleLength`, `numRows` y `tableHeigh`[2]), son todas inmutables, por lo que no deberán ser asignadas de nuevo más adelante, siendo éstas todas las declaraciones que se requieren.

Sin embargo, las variables `hits` (línea 11) y `n` (línea 12) deberán ser asignadas, y sus declaraciones requieren indicar el tipo. Las iteraciones del bucle `for` (línea 14) son controladas por su generador, que en este caso es paralelo, por lo que el bucle será paralelizado. Este código sirve como ejemplo didáctico, aunque es posible realizar algunas optimizaciones, como, por ejemplo, sustituir la actualización atómica de la línea 29 con un incremento que sería automáticamente transformado en una reducción.

1.10. Chapel

`Chapel` [88, 22, 24] es un nuevo lenguaje paralelo de programación que está siendo desarrollado por `Cray Inc.`, en asociación con `CalTech/JPL` como parte de su aportación global al proyecto HPCS.

`Chapel` pretende mejorar la programabilidad de los sistemas paralelos en general, ofreciendo un mayor nivel de expresividad que otros lenguajes paralelos actuales, mejorando la separación entre la expresión de algoritmos y los detalles de la implementación. Para ello, mientras que incorpora muchas de las características de programación moderna, como orientación objetos, también se adquieren algunos compromisos para lograr un alto rendimiento. `Chapel` sigue un modelo de programación paralelo *multithread* de alto nivel, donde el paralelismo no se describe usando un modelo basado en un proceso o tarea, sino en términos de computaciones independientes implementadas usando threads. `Chapel` da soporte a paralelismo de datos, paralelismo de tareas y paralelismo multinivel.

Dos técnicas básicas respaldan la aproximación de `Chapel`, *Multithreading con soporte para localidad* (*Locality Aware MultiThreading*) y la *programación*

[2]en la inicialización de esta variable existe una yuxtaposición de dos "no-funciones". Esto, en notación matemática, indica multiplicación

```
component fortress.executable

export Executable

run(args:String...):()=do

  needleLength = 20          // declaration of
  numRows = 10               // immutable variables
  tableHeight = needleLength numRows

  var hits : RR64 = 0.0  // declaration of mutable
  var n : RR64 = 0.0     // variables of type RR64

  for i <- 1#3000 do       //  3000 iterations
    delta_X = random(2.0) - 1
    delta_Y = random(2.0) - 1
    rsq = delta_X^2 + delta_Y^2

    if 0 < rsq < 1 then
      y1 = tableHeight random(1.0)
      y2 = y1 + needleLength (delta_Y / sqrt(rsq))
      (y_L, y_H) = (y1 MIN y2, y1 MAX y2)

      // increase 'hits' if needle hits fine
      if ceiling(y_L/needleLength) =
                floor(y_H/needleLength) then
        atomic do hits += 1.0 end
      end
      atomic do n += 1.0 end
    end
  end

  probability = hits/n
  pi_est = 2.0/probability
  end
end
```

Listado 1.17: Ejemplo del cálculo de π mediante un código `Fortress`

genérica. La primera técnica es una extensión del modelo PGAS que expresa la relación entre cómputos de un proceso particular y los datos a los que accede. La programación genérica aborda la cuestión de la reusabilidad del código mediante constructos que expresan las cualidades abstractas de las estructuras de datos de forma que los algoritmos pueden entonces ser expresados aceptando cualquier estructura de datos que posea las cualidades requeridas.

Los fundamentos de `Chapel` son una mezcla de las características con más éxito de varios lenguajes importantes de alto nivel, entre los que se destacan `Fortran90/95`, `C` y `C++`, si bien su aproximación al paralelismo está basada preferentemente en lenguajes como `HPF`, `ZPL` y las extensiones `Cray MTA` de `Fortran/C`. Los programadores a los que estos lenguajes les son familiares deberían ser capaces de seguir el modelo de programación de `Chapel`, aunque algunos términos y conceptos pueden guardar algunas diferencias.

`Chapel` es un lenguaje orientado a objetos que soporta herencia y sobrecarga y en el que se pueden destacar algunas características como tipos estructurados que pueden definirse junto con constructores y funciones asociados a ellos, una sentencia `use` que puede ser utilizada para controlar el acceso a los campos de los tipos estructurados, tipos `union`, etc.

Un tipo de alto nivel a destacar es la *secuencia* (`sequence`), cuyo uso ofrece realizar iteraciones sobre conjuntos de un modo abstracto, relegando los detalles de la implementación al compilador o al sistema. Una secuencia es una lista de expresiones, todas ellas con el mismo tipo y pueden ser usadas para controlar bucles `for` o iteradores similares.

Los programadores de la mayoría de lenguajes de alto nivel están familiarizados con el concepto de datos agregados. `Chapel` extiende esta idea introduciendo unos objetos de alto nivel llamados *dominios* (*domains*). Éstos no son datos agregados en sí mismos, sino descripciones de cómo tales datos son direccionados. Por ejemplo, un vector convencional en `Fortran` o `C` es accedido mediante un conjunto de enteros, tantos como requiera el número de dimensiones del vector, estando establecidos unos ciertos límites para estos índices. De esta forma, en estos lenguajes el dominio está especificado mediante uno o varios enteros limitados. Sin embargo, en `Chapel` los dominios no están restringidos a enteros, sino que para direccionar pueden utilizarse cadenas, booleanos, tipos enumerados o cualquier otro tipo escalar. En `Chapel` se dispone de métodos para gestionar estos índices, tanto para añadirlos como para eliminarlos del dominio, así como para determinar si un índice específico ha sido o no definido dentro de un dominio.

El listado 1.18 muestra una implementación del algoritmo de Jacobi

realizado en `Chapel`. El tamaño de la cuadrícula para los cálculos se define usando dos dominios (lineas 6 y 8), y el espacio del problema se declara a través de estos dominios. Las direcciones en la cuadrícula se representan usando t-uplas de dos valores (líneas 14 y 15), que se añaden a cualquier coordenada de la cuadrícula (líneas 19 y 20) para indicar los puntos correctos que rodean la coordenada actual. En la línea 18 se muestra el bucle paralelo `forall`, que realiza los principales cálculos para todas las coordenadas ij dentro del espacio del problema de forma concurrente. Finalmente, se realiza una operación de reducción (líneas 21 y 22) para calcular la diferencia entre el viejo espacio del problema y el nuevo.

```
1  config var n = 5,               // size of n x n grid
2             epsilon = 0.00001;   // convergence tolerance
3
4  def main() {
5          // domain for grid points
6    const ProblemSpace = [1..n, 1..n],
7          // domain including boundary points
8          BigDomain = [0..n+1, 0..n+1];
9
10   var X, XNew: [BigDomain] real = 0.0;
11   X[n+1, 1..n] = 1.0;
12   var iteration = 0, delta: real;  // mensure of converg.
13
14   const north = (-1,0), south = (1,0),
15         east  = (0,1),  west = (0,-1);
16
17   do {
18     forall ij in ProblemSpace do
19       XNew(ij) = (X(ij+north) + X(ij+south) +
20                   X(ij+east)  + X(ij+west)) / 4.0;
21     delta = max reduce abs(XNew[Problem5pace] -
22                            X[Problem5pace]);
23     X[ProblemSpace] = XNew[ProblemSpace];
24     iteration += 1;
25
26   } while (delta > epsilon);
27 }
```

Listado 1.18: Implementación en `Chapel` del algoritmo de Jacobi

1.11. X10

X10 [154, 30] es un nuevo lenguaje experimental que actualmente se encuentra bajo desarrollo por IBM y algunos socios académicos, dentro del programa HPCS. Su nombre proviene de la intención de sus diseñadores de crear, sobre 2010, un lenguaje que sea diez veces más productivo que los lenguajes y librerías normalmente usados en la actualidad para entornos de CAP.

Entre sus objetivos principales se encuentran el de desarrollar un lenguaje de alto nivel que sea más productivo y que pueda soportar mayores niveles de abstracción que otros lenguajes actuales, así como que pueda explotar múltiples niveles de paralelismo y accesos no uniformes de datos que son críticos para obtener rendimiento escalable en sistemas CAP actuales y futuros.

X10 pretende contribuir a la mejora de la productividad desarrollando un nuevo modelo de programación, combinado con un nuevo conjunto de herramientas que se integran dentro de Eclipse y unas nuevas técnicas de implementación para desarrollar un paralelismo optimizado y escalable. X10, en palabras de sus desarrolladores, es un lenguaje moderno, paralelo, seguro, escalable, de código flexible, orientado a objetos distribuidos. Para ello, en lugar de diseñar totalmente un nuevo lenguaje con su sintaxis, tipos, estructura de datos y paralelismo, X10 se define como una extensión de Java, pensado para ser fácilmente accesible para aquellos programadores de este lenguaje.

Esto representa un intento de migrar un lenguaje existente al campo de la computación de alta productividad y rendimiento, de un modo similar al que siguen otras aproximaciones como Co-Array Fortran, UPC y Titanium. Sin embargo, mientras que estos tres lenguajes usan un modelo PGAS para expresar el paralelismo, X10 lo extiende a un modelo *GALS* (***G**lobally **A**synchronous, **L**ocally **S**ynchronous*) [29].

Los elementos básicos de X10 siguen el paradigma orientado a objetos de Fortress y Chapel. Los bloques básicos que construyen los programas de X10 son las interfaces genéricas y las clases. Las clases escalares poseen campos, métodos y tipos internos, y pueden heredar atributos estableciendo como subclase otra clase. Sin embargo, X10 no soporta las declaraciones de clases de vectores, lo que significa que un usuario no puede definir nuevos tipos para estas clases, creándose como instancias de tipos de vectores a través de constructores de tipo vector. Además, X10 no tiene operación de derreferencia y no soporta aritmética de punteros, lo cual se traduce en códigos mucho más

seguros.

Se dice que `X10` posee un sistema de tipos seguros –*type safe*–, (se garantiza que una ubicación contiene valores legítimos para el tipo que se refiere a ella), *memory safe* (los accesos son comprobados dinámicamente y automáticamente para asegurar que estén dentro de los límites permitidos) y *pointer safe* (valores que no pueden ser nulos no podrán lanzar excepciones de punteros nulos). Además, también es *place safe* (siguiendo la nomenclatura de `X10` que describe un *lugar* como la localidad de datos y actividad) y, si sólo se usan relojes y secciones atómicas incondicionales, se garantiza que no habrá interbloqueos.

El listado 1.19 recoge la implementación en `X10` del mismo algoritmo de Jacobi que se mostró en el listado 1.18 (página 45). En este caso, en vez de usar dominios, el tamaño del problema se define por medio de regiones y distribuciones (líneas 6 a 11). La paralelización de los cálculos se consigue implícitamente para cada punto $[i, j]$ a través de la asignación a una distribución (líneas 22 a 26).

1.12. `llc`

Hemos querido finalizar este capítulo con un breve apartado que incluya a `llc` en esta revisión. El resto de capítulos de esta memoria están dedicados a explicar en profundidad las características y detalles de `llc`, por lo que aquí sólo presentaremos un ejemplo de una aplicación implementada en `llc`, de forma que le permita al lector comparar nuestro estilo de programación con el del resto de aproximaciones estudiados en este capítulo.

Como ejemplo mostramos el listado 1.20 que recoge la versión en nuestro lenguaje del algoritmo paralelo de cálculo de π que ha sido presentado con anterioridad para otros lenguajes. Este código conserva la estructura del programa homólogo secuencial mostrado en el listado 1.1 (página 4). Si lo comparamos con la versión `OpenMP` que se muestra en el listado 1.2 (página 5) podremos observar cómo la única diferencia con éste es la inclusión de la directiva de `llc` `reduction_type` (línea 8). Sólo con esta modificación sobre la versión de `OpenMP`, nuestro compilador es capaz de generar un código eficiente de rendimiento similar al obtenido en el código *ad-hoc* de `MPI` recogido en el listado 1.3 (página 8).

```
1  public class Jacobi extends x10Test {
2
3    const int N = 5;
4    const double epsilon = 0.0001;
5    const double epsilon2 0.000000001;
6    const region(:rank==2) RInner = [1:N, 1:N];
7    const region(:rank==2) R = [0:N+1, 0:N+1];
8    const dist(:rank==2) D = (dist(:rank==2))
9                              dist.factory.block(R);
10   const dist(:rank==2) DInner = D | RInner;
11   const dist(:rank==2) DBoundary = D - RInner;
12   const int EXPECTED_ITERS = 97;
13   const double EXPECTED_ERR = 0.0018673382039402497;
14
15   final double[.] XNew = new double[D] (point p[i,j]){
16     return DBoundary.contains(p) ? (N-1)/2 : N*(i-1)+(j-1);
17   };
18
19   public boolean run() {
20     int iters = 0;
21     double err;
22     while (true) {
23       final double[:distribution==this.DInner] X =
24                      new double[DInner] (point [i,j]){
25         return (XNew[i+1,j] + XNew[i-1,j] +
26                 XNew[i,j+1] + XNew[i,j-1]) / 4.0;
27       };
28       if ((err =
29            ((XNew | this.DInner)-X).abs().sum()) < epsilon)
30         break;
31       XNew.update(X);
32       iters++;
33     }
34     return Math.abs(err - EXPECTED_ERR) < epsilon2
35                                 && iters == EXPECTED_ITERS;
36   }
37
38   public static void main(String[] args) {
39     new Jacobi().execute();
40   }
41 }
```

Listado 1.19: Implementación en X10 del algoritmo de Jacobi

```
1   w = 1.0 / N;

3 #pragma omp parallel private(i, local)
4   {
5 #pragma omp single
6     pi_llc = 0.0;
7 #pragma omp for reduction (+: pi_llc)
8 #pragma llc reduction_type (double)
9     for (i = 0; i < N; i++) {
10      local = (i + 0.5)*w;
11      pi_llc = pi_llc + 4.0/(1.0 + local*local);
12    }
13  }
14  pi_llc *= w;
```

Listado 1.20: Ejemplo del cálculo de π mediante un código `llc`

Capítulo 2

El Lenguaje `llc`

2.1. Introducción

El trabajo que hemos desarrollado tiene como idea central el diseño de un lenguaje paralelo independiente de la arquitectura, que permita el desarrollo de aplicaciones eficientes y que sea de fácil uso para una comunidad científica no especializada en paralelismo, entre otras características.

En este capítulo nos centraremos en la descripción del lenguaje paralelo que hemos desarrollado y que hemos denominado `llc` (**L**a **L**aguna **C**) [56]. A continuación se detallan las características que definen al lenguaje.

- *Portabilidad*: En el diseño de `llc` hemos enfatizado la *portabilidad*, imponiendo como característica principal que el lenguaje debe ser independiente de la arquitectura y sistema paralelo en el que se vayan a desarrollar las aplicaciones.
- *Usabilidad*: `llc` debe ser *usable* en el sentido que debe ser fácil de aprender y utilizar, sobre todo para los usuarios a los que va preferentemente dirigido: científicos que poseen ciertas nociones de programación, pero que no necesariamente son expertos en paralelismo.
- *`llc` como extensión de un lenguaje base*: Las dos mencionadas características de *portabilidad* y *usabilidad* nos han llevado a desechar la idea del diseño de `llc` como un nuevo lenguaje, ya que supondría un obstáculo para los usuarios, que se verían forzados a aprender un nuevo lenguaje.

  Por ello `llc` se articula como extensión del lenguaje `ANSI C`, que es en la actualidad uno de los más extendidos y usados en contextos científicos.

- *No interferencia con el código original*: `llc` se sirve de directivas de compilador (*pragmas*) para expresar paralelismo. Estas directivas no rompen la estructura del código fuente previo, lo que permite conservar la funcionalidad del programa original al extenderlo mediante directivas `llc`.

- *Elección de `OpenMP` como lenguaje referencia*: Para expresar paralelismo, `llc` utiliza un planteamiento similar al empleado por `OpenMP`. Mediante esta aproximación, al diseñar `llc` apostamos por un modelo de programación paralela que resulte simple al usuario no experto. Por otro parte, la aproximación elegida permite un fácil uso de `llc` por la amplia comunidad de usuarios de `OpenMP`.

- *Compatibilidad con OpenMP*: Hemos realizado el diseño de los constructos y directivas de `llc` de modo que se asegurara la compatibilidad con el estándar 2.5 de `OpenMP` [123] tanto como fuese posible.

- *Minimización de directivas*: `llc` se presenta como un lenguaje completo que dispone de sus propias directivas para expresar paralelismo. Sin embargo, si el usuario toma como partida un código válido `OpenMP`, el conjunto de directivas `llc` que es necesario añadir es, en general, reducido debido a la característica de compatibilidad de `llc` con `OpenMP`. Esta propiedad propicia que un código válido de `OpenMP` sea un buen punto de partida para desarrollar una aplicación `llc`.

- *Paralelización incremental*: `llc` permite que la paralelización de los códigos secuenciales pueda realizarse de forma incremental, dando la posibilidad al usuario de marcar o desmarcar mediante el uso de directivas qué regiones del código son susceptibles de ser paralelizadas, en el orden que considere oportuno.

- *Basado en el modelo* OTOSP: `llc` está basado en el modelo de computación colectiva OTOSP, que será presentado en el apartado 2.2 (página 53).

- *Compilador*: `llc` se ha diseñado de forma que sea factible un compilador que dé soporte a este lenguaje y que produzca como salida programas paralelos eficientes y portables. Este compilador ha sido diseñado e implementado y será presentado en el capítulo 3 de esta memoria.

- *Código fuente multifuncional*: Otra ventaja adicional de la compatibilidad de `llc` con `OpenMP` es que a partir de un mismo

código fuente válido `OpenMP` anotado con las directivas de `llc` se puedan obtener tres versiones ejecutables para diferentes fines, únicamente variando el compilador usado:

- Binario secuencial: si usamos un compilador secuencial que ignore todas las directivas (`#pragma`), tanto de `OpenMP` como de `llc`, el resultado será un programa secuencial.
- Binario paralelo `OpenMP`: si es usado un compilador de `OpenMP` que ignore las directivas de `llc`, se obtendrá un programa paralelo `OpenMP` válido únicamente en máquinas de memoria compartida.
- Binario paralelo `MPI`: por último, tras procesar el código fuente con el compilador de `llc` y generar el ejecutable enlazando con librerías de `MPI` (véase figura 3.1 en página 106), el resultado final será un código paralelo `MPI` que será válido tanto para máquinas de memoria compartida como distribuida.

2.2. El Modelo *OTOSP*

`llc` se basa en el modelo de computación colectiva denominado OTOSP (***One*** ***T****hread is* ***O****ne* ***S****et of* ***P****rocessors*) [44]. Para definir este modelo abstracto, en primer lugar se considerará que disponemos de un sistema paralelo con un número arbitrario de procesadores. Este conjunto de procesadores está interconectado por medio de una red de cualquier topología que permita la comunicación entre cualquier par de procesadores. A su vez, consideraremos que este sistema paralelo es, en general, de memoria distribuida y que cada procesador estará dotado de su propia memoria privada.

En el modelo de computación colectiva todos los procesadores del sistema están organizados en conjuntos formados por al menos un procesador, de modo que cada uno de ellos deberá pertenecer en todo momento a uno y sólo a uno de estos conjuntos. Un conjunto se caracteriza porque todos los procesadores que lo forman tienen la misma visión de la memoria y ejecutan el mismo programa o tarea sobre los mismos datos, de modo que replican el cómputo.

Estos procesadores se diferencian dentro de su grupo mediante un identificador único. Al comienzo de la ejecución del programa sólo existe un conjunto de procesadores, denominado *conjunto raíz*, al que pertenecen todos los procesadores de la máquina.

Por definición, cualquier conjunto de procesadores sólo puede realizar tres tipos de operaciones en el modelo OTOSP:

1. Cualquier cómputo secuencial: asignaciones, bucles, llamadas a función, etc.
2. Funciones de división.
3. Operaciones colectivas.

Mientras se estén ejecutando cómputos secuenciales, el conjunto de procesadores no sufre modificaciones y todos los procesadores que pertenecen al mismo ejecutarán las mismas instrucciones secuenciales sobre los mismos datos.

Sin embargo, al finalizar los cómputos secuenciales y encontrar una región de código paralelo, se realiza una *operación de división* sobre el conjunto de procesadores. Las funciones de división deberán ser ejecutadas por todos los procesadores del conjunto afectado y, de esta forma, el conjunto inicial se divide en tantos subconjuntos como ejecuciones simultáneas produzca la región paralela que ha generado la operación de división. La decisión de cuántos procesadores del conjunto inicial se asignan a cada subconjunto se lleva a cabo según criterios de equilibrado de carga, asegurando siempre que cada subconjunto dispone al menos de un procesador.

Si dentro de esta región paralela se encuentra, a su vez, otra región paralela, se volverá a producir otra operación de división, generándose nuevos subconjuntos en un nivel inferior y dando lugar a paralelismo anidado. Para una máquina ideal con infinitos procesadores, esta operación de subdivisión se puede repetir tantas veces como regiones paralelas se encuentren en el código que ejecuta cada subconjunto. Sin embargo, en una máquina real con un número finito de procesadores, se tiene que considerar la situación en la cual un conjunto llegue al subconjunto mínimo formado por un único procesador. En este caso, no es posible seguir realizando más funciones de división, ya que el subconjunto sólo dispone de un procesador, por lo que si se encontraran más regiones paralelas, éstas deberán ser ejecutadas de forma secuencial por el único procesador disponible.

Este procedimiento de operaciones de división produce como resultado una *jerarquía de conjuntos* que, como muestra la figura 2.1, puede ser representada por un árbol. En dicha figura los conjuntos y subconjuntos de procesadores están representados por los nodos del árbol y los puntos dentro de cada nodo indican el número de los procesadores que pertenecen a cada conjunto.

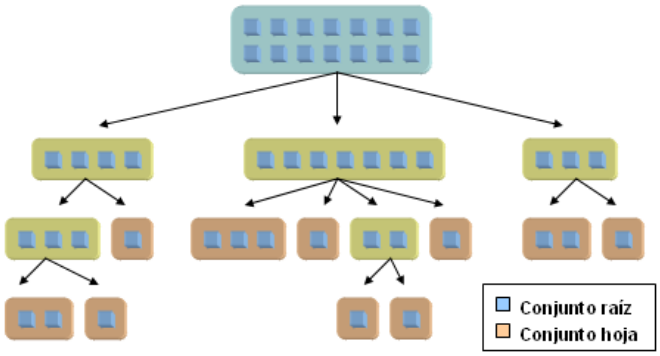

Figura 2.1: Estructura jerárquica de procesadores en el modelo OTOSP.

Los nodos marcados en rojo en la figura 2.1 corresponden a los conjuntos de procesadores de creación más reciente y que se encuentran activos en un determinado instante de cómputo. Estos conjuntos se denominan *conjuntos hoja* y, en cualquier instante de la ejecución del programa, todo procesador pertenece a uno de estos conjuntos hoja, siendo el conjunto raíz el único conjunto hoja existente al iniciar la ejecución del programa.

Para cualquier instante de la ejecución del programa, los nodos de la jerarquía representan a conjuntos donde todos los procesadores ejecutan la misma tarea sobre los mismos datos. Los conjuntos hoja representan a tareas que están siendo ejecutadas en ese preciso instante, mientras que los nodos interiores corresponden con tareas cuya finalización está a la espera de que se completen todas las subtareas generadas por dicho nodo.

La finalización de todas las subtareas que generó la división de un nodo da lugar al tercer tipo de operaciones del modelo: *operaciones colectivas*. Los subconjuntos hijos se originan durante la aplicación de una función de división como resultado de la entrada en una región paralela y, al terminar dicha región paralela, estos subconjuntos hijos se combinan mediante las denominadas operaciones colectivas, para volver a restituir el conjunto padre que los originó.

Una de las condiciones que deben cumplir todos los procesadores que pertenecen a un mismo conjunto es el de poseer la misma visión de memoria. Como cada subconjunto corresponde con una tarea diferente y cada uno de ellos ha realizado cómputos distintos obteniendo sus propios resultados, es necesario que los procesadores de cada subconjunto comuniquen los resultados obtenidos a los procesadores del resto de conjuntos, con el fin de poder reintegrarse en el grupo original.

Esto conlleva que las operaciones colectivas lleven implícito un proceso de comunicación de todos los resultados obtenidos por cada subtarea. A su vez, durante las operaciones colectivas se produce un punto de sincronización en la ejecución del programa, ya que los procesadores de los conjunto hoja no prosiguen la ejecución hasta que todos los demás procesadores de los conjuntos hoja afectados hayan finalizado y los resultados hayan sido compartidos.

Según este modelo, independientemente del programa que se ejecute y los cómputos que se realicen, tanto sean secuenciales como paralelos, la ejecución siempre comienza con un conjunto raíz que engloba a todos los procesadores disponibles y que generará todos los subconjuntos tras aplicar las funciones de división. A su vez, la ejecución siempre finalizará dando como resultado el mismo conjunto raíz inicial al finalizar todas las operaciones colectivas.

Si trasladamos este modelo teórico a un caso práctico real implementado usando `MPI`, podemos utilizar cualquier máquina paralela de las presentadas en el apéndice A para representar el sistema con un número acotado de procesadores. A pesar de que el modelo está concebido sobre máquinas de memoria distribuida, es también válido para máquinas de memoria compartida, ya que `MPI` gestiona partes de esta memoria compartida como privada para cada procesador. En cuanto a las funciones de división, éstas se corresponden con las funciones `MPI` de división de comunicadores (`MPI_Comm_split()`, `MPI_Cart_sub()`, etc.) y las operaciones colectivas estarían representadas por las funciones colectivas de `MPI` (`MPI_Reduce()`, `MPI_Allreduce()`, etc.) aplicadas en el ámbito de un comunicador, o bien por operaciones de envío y recepción de datos (`MPI_Send()`, `MPI_Recv()`, etc.), siempre que el esquema de comunicaciones implique una comunicación colectiva.

2.3. Constructos y directivas en `llc`

En `llc` el paralelismo se expresa mediante directivas de compilador. Una de sus principales ventajas es que no rompen la estructura del código original, conservando su funcionalidad y permitiendo a su vez la portabilidad del mismo. Estas directivas están formadas por la palabra reservada `pragma`, a la que generalmente sigue un identificador (`id`). El uso de este identificador permite especificar qué compilador o compiladores deberán interpretar la información indicada en la directiva. En general, un compilador ignora aquellas directivas `pragma` cuyo contenido no reconoce.

Los compiladores de `OpenMP` interpretan aquellas directivas con identificador `omp`, aunque también existen algunas directivas que son específicas para el compilador `intel` de `OpenMP`, usándose en estos casos el identificador `intel omp`. Por su parte, las directivas y constructos de `llc` se distinguen mediante el identificador `llc`. Dada la compatibilidad con `OpenMP`, en `llc` se permite el uso indistinto de estos tres identificadores, por lo que el compilador de `llc` analizará cualquier directiva que tenga como identificador alguno de los siguientes:

```
llc | omp | intel omp
```

A pesar de que el programador puede usar libremente estos identificadores en una aplicación `llc`, por convenio y para una mayor compatibilidad con `OpenMP` y claridad del código, se recomienda el uso del identificador que corresponde con el lenguaje al que pertenece la directiva o constructo en cuestión. En esta memoria se seguirá este convenio.

Las directivas de compilador acaban en el primer carácter de nueva línea (aquí se representará como `nl`) que se encuentra en la línea de la directiva. Sin embargo, es posible escribir directivas de más de una línea usando para ello la barra invertida (\) inmediatamente antes del fin de línea, tantas veces como se desee.

Las directivas son insertadas por el usuario en aquellas zonas susceptibles de ser paralelizadas. El objetivo de estas directivas en `llc` es el indicar al compilador la siguiente información:

- *Tipo de paralelismo*: El usuario es el responsable de indicar el paradigma paralelo que el compilador debe seguir para generar el código final. Ejemplos de estos modelos pueden ser la paralelización de bucles, secciones, pipelines, colas de tarea, etc.

- *Parámetros*: El compilador debe recibir otros parámetros y modificadores, como información de control sobre los atributos de las variables que indiquen cuáles de ellas son privadas y cuáles compartidas, qué resultados deben ser comunicados, etc.

En `OpenMP`, el usuario especifica en una misma directiva el tipo de paralelismo, que indica mediante el *nombre de la directiva*, y los parámetros,

que especifica mediante *cláusulas*. A continuación se presenta la sintaxis general de las directivas en `OpenMP`[1]:

```
#pragma omp nombre-directiva {cláusula}* nl
```

Tanto los nombres de directivas como las cláusulas son sensibles a mayúsculas y minúsculas, escribiéndose generalmente siempre en minúsculas. Cada directiva admite un determinado conjunto de cláusulas. Estas cláusulas pueden ser escritas en cualquier orden dentro de la directiva y, en general, aceptan una lista de parámetros encerrados entre paréntesis y separados por comas.

La sintaxis de `llc` difiere ligeramente de la de `OpenMP`. Para dotar de más claridad al código, el tipo de paralelismo o constructo paralelo y las cláusulas o parámetros se especifican usando directivas independientes. De este modo, en `llc` disponemos de dos tipos de directivas, por un lado las *directivas-constructos* (que llamaremos *constructos*) y las *directivas-cláusulas*(que en `llc` denominaremos simplemente *directivas*).

La sintaxis general de los constructos y directivas que contiene una aplicación `llc` difiere según el camino que haya seguido el programador para desarrollar el código. Usualmente nos encontramos con las siguientes dos alternativas, o una combinación de ellas:

1. Se parte de un código secuencial y se utilizan únicamente directivas de `llc` para especificar el paralelismo. En este caso la sintaxis general es la de las directivas `llc`:

   ```
   #pragma llc directiva-constructo-llc nl
   {#pragma llc directiva-cláusula-llc nl}*
   ```

2. Se toma como punto de partida un código válido `OpenMP`. En este caso es posible conservar sus directivas y completarlas usando directivas `llc` si fuese necesario. Esta forma de programar permite obtener un programa `llc` que conserva también la validez de un programa `OpenMP` original. La sintaxis general de las directivas en este caso es:

   ```
   #pragma omp directiva-constructo-omp {cláusula-omp}* nl
   {#pragma llc directiva-cláusula-llc}* nl
   ```

[1]En esta memoria utilizaremos la notación *EBNF* para indicar la sintaxis de las directivas.

La necesidad de añadir directivas `llc` a un programa original `OpenMP` se debe a que sus constructos y cláusulas no aportan toda la información necesaria para implementar el modelo OTOSP. `OpenMP` es un lenguaje para máquinas de memoria compartida y, por defecto, todas las variables son compartidas. Este hecho determina que una mayoría de cláusulas de `OpenMP` estén dedicadas a especificar el ámbito privado de las variables que lo requieran.

Sin embargo, en `llc` se produce la situación opuesta, las variables son privadas por defecto y se requieren directivas que indiquen información sobre variables compartidas, siendo pocas las cláusulas de `OpenMP` diseñadas con este fin. Por este motivo, `llc` ha añadido un pequeño conjunto de directivas que cubren esta necesidad y permiten especificar el carácter compartido de las variables cuando se programa en `llc`.

Otra decisión que se ha tomado para aumentar la compatibilidad de `llc` con `OpenMP` es permitir una dualidad en la sintaxis de las directivas-cláusulas comunes. De este modo, en caso de existir ambas posibilidades, el usuario puede elegir al indicar una cláusula si desea usar la sintaxis de `OpenMP` o la de `llc`.

Por otro lado, también hay que indicar que no todas las directivas de `OpenMP` tienen sentido en el modelo OTOSP. Para más información sobre la compatibilidad entre las directivas de `llc` y `OpenMP`, el lector puede consultar el apartado 2.14 (página 97), donde se recogen varias tablas de compatibilidad entre ambos lenguajes.

Las directivas de `llc` también son sensibles a mayúsculas y minúsculas, escribiéndose generalmente con minúsculas. La *directiva-constructo* siempre debe escribirse en primer lugar, escribiendo a continuación las *directivas-cláusulas*. En estas últimas, en general, el orden en que se especifiquen no es determinante[2].

Al igual que otros lenguajes, como `OpenMP`, `llc` da soporte únicamente a la paralelización especificada directamente por el usuario, que es quien indica el tipo de acciones que el compilador debe emplear para transformar el código fuente anotado en un programa paralelo de salida. Sin embargo, `llc` ha sido diseñado para minimizar en lo posible las acciones que debe realizar el usuario sobre el código de origen.

No obstante, es labor y responsabilidad del usuario indicar de forma correcta las directivas de constructos y cláusulas de `llc` y desarrollar una

[2]En esta memoria se indicará de forma explícita los pocos casos en que deben seguir un orden específico

aplicación conforme con `llc`, es decir, una aplicación que sigue todas las reglas y restricciones impuestas por el lenguaje. De este modo, el usuario deberá evitar situaciones tales como dependencias, conflictos, bloqueos y otros problemas que pueden dar lugar a programas no conformes con `llc`.

El modelo OTOSP en el que se basa `llc` permite, en general, *la anidación de constructos*, posibilitando una implementación eficiente de paralelismo multinivel. En las secciones siguientes realizaremos una revisión detallada de los constructos de `llc` y las directivas que admite cada constructo.

2.4. Constructo `parallel`

En `OpenMP` el constructo `parallel` se usa para indicar una región de código que será ejecutada en paralelo. Para la ejecución de esta región se crea un equipo de threads y cada thread ejecutará el código delimitado por el constructo `parallel`, modificándose las condiciones en las que se lleva a cabo esta ejecución mediante el uso de cláusulas y/o otros constructos dentro de la región paralela.

2.4.1. Ámbito de aplicación del constructo `parallel` en `llc`

En `llc` este constructo no tiene especial relevancia, ya que el modelo OTOSP contempla que todos los procesadores del grupo deben ejecutar en paralelo el código de la aplicación, lo cual supone algo similar a un constructo `parallel` que englobara todo el código.

Sin embargo, por compatibilidad con `OpenMP`, `llc` admite este constructo, realizando la comprobación adicional que los diferentes constructos que deben estar dentro de una región paralela en `OpenMP` lo estén también en un programa `llc`, emitiendo un aviso (no un error) si no es así. La sintaxis de este constructo en `llc` es la siguiente:

```
#pragma omp parallel {omp_parallel_clauses}*
```

2.5. Constructo `for`

El constructo `for` es uno de los más ampliamente usados. Este constructo da soporte a la paralelización de bucles, siguiendo un modelo *forall* [95]. Dadas las características de la mayoría de aplicaciones científicas, es muy común que que los algoritmos implementados contengan un bucle que aplica una serie de operaciones de forma repetitiva sobre un conjunto cambiante de datos.

En estos casos, una forma natural de paralelizar estos códigos consiste en distribuir la ejecución de las iteraciones del bucle entre los procesadores disponibles. Para ello, se deben cumplir una serie de condiciones, como, por ejemplo, no deben existir dependencias *loop-carried* (dependencias originadas por el bucle) [91] ni conflictos entre las distintas iteraciones. Si estas condiciones se cumplen, entonces es posible paralelizar el bucle siguiendo un modelo *forall*.

2.5.1. Ámbito de aplicación del constructo `for` en `llc`

`llc` permite la paralelización de bucles `for` mediante el uso de la siguiente directiva de compilador:

```
#pragma omp [parallel] for {omp_for_clauses}*
```

El constructo `for` debe estar en el ámbito en una región paralela para su correcto funcionamiento. El uso de la palabra reservada `parallel` en la directiva depende de si la región paralela ha sido declarada con anterioridad (en este caso se omite) o bien la región paralela se está declarando con esta directiva (debe usarse esta palabra reservada).

2.5.1.1. Paralización de *bucles generales* `for` en `llc`

Una restricción que el estándar 2.5 de `OpenMP` [123] impone a los bucles `for` a paralelizar es que tienen que estar expresados en su *forma canónica*, ya que el cálculo de número de iteraciones se produce a la entrada del bucle.

Si no se cumple esta condición, el estándar indica que se puede obtener un comportamiento indeterminado. Por otro lado, en `OpenMP` la variable que sirve de índice al bucle debe ser obligatoriamente de tipo entero con signo.

Estas restricciones imposibilitan la paralelización de bucles en `OpenMP` cuando, a pesar de tener un número fijo de iteraciones, éste valor no puede ser calculado a la entrada del bucle, o bien la variable índice no es de tipo entero.

El listado 2.1 recoge un ejemplo de esta situación. En este listado se tiene un bucle que recorre una lista de punteros, donde la variable `p` es un puntero y la función `f` realiza cualquier operación sobre los datos del puntero, pero sin alterar la lista. En este caso, el número de iteraciones no puede ser fácilmente calculado como una operación aritmética. Por otro lado, la variable índice es un puntero, otra razón por la que `OpenMP` no podrá llevar a cabo la paralelización de este código.

```
1  for (p = pini; p = p->next; p != NULL) {
2    f(p);
3  }
```

Listado 2.1: Bucle general

En `llc` se han contemplado situaciones como la descrita y se permiten bucles `for` que no estén en forma canónica y/o la variable índice no sea de tipo entero con signo. Estos bucles los denominamos *bucles generales* y se permite su paralelización, siempre que se asegure que el número de iteraciones permanezca constante durante la ejecución del bucle, si bien no es necesario que pueda calcularse a priori.

2.5.1.2. Restricciones

Cualquier bucle `for` que cumpla las siguientes restricciones puede ser paralelizado en `llc`:

- No deben existir dependencias del tipo *loop-carried* entre las iteraciones, esto es, el orden en que se ejecuten las mismas no debe influir en el resultado obtenido
- El número de iteraciones debe ser fijo, aunque no es necesario que éste pueda ser calculado a priori ni que el bucle `for` esté representado en su forma canónica
- El código del bucle debe ser un bloque estructurado que no debe contener sentencias de ruptura de control (`break`, `goto`, ...).

2.5.2. Directivas del constructo `for`

2.5.2.1. Directiva `result`

La directiva `result` de `llc` se utiliza con el fin de comunicar los resultados obtenidos por un subgrupo de procesadores al resto de subgrupos. Esta operación colectiva es utilizada con el fin de garantizar la consistencia de memoria, usándose para comunicar datos que están organizados en *bloques contiguos* de memoria.

La sintaxis de esta directiva es la siguiente:

```
#pragma llc result (p, n {, p, n}*)
```

1. `p`: puntero al principio de la zona de memoria a comunicar.
2. `n`: número de elementos a comunicar a partir de la zona de memoria indicada por `p`.

El número de elementos (`n`) corresponde al tipo de datos apuntado por `p`, no siendo necesario especificar la cantidad de memoria que éstos ocupan, ya que será el compilador quien lo calcule. Si se desea especificar más de una región de memoria a comunicar, se puede optar por incluir tantos pares `p, n` como se desee, usar más de una directiva `result` o bien una combinación de ambas posibilidades.

```
1  #pragma omp parallel for
2  #pragma llc result (&v1[i], 1, &v2[i][0], N)
3  for (i = 0; i < N; i++) {
4    v1[i] = f(i);
5    for (j = 0; j < N; j++) {
6      v2[i][j] = g(i, j);
7    }
8  }
```

Listado 2.2: Directiva `result` en accesos contiguos a memoria

En el listado 2.2 se muestra un ejemplo de uso de la directiva `result`. En este código vemos como los vectores `v1` y `v2` son modificados, por lo que los espacios de memoria escritos deben comunicarse al resto de subgrupos de

procesadores antes de poder reunirlos en el grupo original. En cada iteración sólo se escribe un elemento del vector `v1`, mientras que del vector `v2` se escriben `N` elementos. En ambos casos, se trata de accesos contiguos de memoria, por lo que la directiva a utilizar es `result`, tal y como se indica en la línea 2.

El código anterior también sirve de ejemplo para mostrar que existen varias alternativas válidas para marcar los bloques de memoria. En el caso de `v2` podemos expresarlo indicando que se escriben `N` elementos empezando en la posición `&v2[i][0]`, o bien se podría haber indicado que se escribía un solo elemento a partir del puntero `v2[i]`. Ambas formas de expresar el bloque de memoria a comunicar son equivalentes y producirán los mismos resultados. Por otro lado, en el listado 2.2 se ha utilizado una única directiva `result` con dos listas de pares de elementos, lo cual también podría haber sido expresado con dos directivas, conteniendo un par de elementos cada una.

La directiva `result` lleva implícita una sincronización de todos los procesadores del grupo al finalizar la ejecución del bucle paralelo.

2.5.2.2. Directiva `rnc_result`

En algunas situaciones, las zonas de memoria accedidas para modificación no están contiguas en memoria, pero siguen un patrón regular de acceso. Un ejemplo de este tipo de acceso lo encontramos cuando una matriz se representa mediante un vector y se realizan accesos de escritura en una submatriz. En estos casos, al no tratarse de un acceso contiguo, no puede utilizarse la directiva `result` anteriormente descrita, sino que debe optarse por la directiva `rnc_result` (*r*`egular` *n*`on-c`*o*`ntiguous result`). La sintaxis de esta directiva se describe a continuación y consiste en una lista de cuartetos.

```
#pragma llc rnc_result (p, n, m, r {, p, n, m, r}*)
```

1. `p`: puntero al principio de la zona de memoria a comunicar.
2. `n`: número de elementos a comunicar a partir de la dirección especificada por `p`.
3. `m`: número de elementos que no serán comunicados a partir de los `n` comunicados.

4. `r`: número de veces que se repetirá el patrón de comunicación descrito.

```
1  int v[N * N];

3  #pragma omp parallel for
4  #pragma llc rnc_result (&v[0], N/B, (N-(int)(N/B)), N/A)
5  for (i = 0; i < N/A; i++) {
6    for (j = 0; j < N/B; j++) {
7      v2[i * N + j] = g(i, j);
8    }
9  }
```

Listado 2.3: Directiva `rnc_result` en accesos regulares no contiguos a memoria

El listado 2.3 muestra un ejemplo de uso de esta directiva. En este ejemplo se asume que todas las constantes (`N`, `A`, `B` y `C`) son números enteros no negativos. El acceso que se produce al vector `v` no es contiguo, pero sigue un patrón regular. Al analizar el código se puede observar cómo se accede a los primeros `N/B` elementos del vector `v`, y luego se produce un salto de `N - N/B` elementos. Este patrón se repite tantas veces como iteraciones tiene el primer bucle, es decir, en `N/A` ocasiones. Esta información es la que se le especifica a la directiva `rnc_result` de la línea 4.

La directiva `rnc_result` lleva implícita una sincronización de todos los procesadores del grupo al finalizar la ejecución del bucle paralelo.

2.5.2.3. Directiva `nc_result`

Hemos visto cómo es posible comunicar regiones de memoria cuando éstas son contiguas o bien siguen un patrón regular. Sin embargo, en algunas ocasiones los accesos a memoria son aleatorios o no se rigen por una patrón fijo. Para estas situaciones, `llc` dispone de la directiva `nc_result` (*non-contiguous* `result`). La sintaxis completa de esta directiva es la siguiente:

```
#pragma llc nc_result (p, n, v {, p, n, v}*)
```

1. `p`: puntero al principio de la zona de memoria a comunicar.

2. `n`: número de elementos a comunicar a partir de la dirección indicada por `p`.

3. `v`: puntero al principio de la variable accedida.

A diferencia que el resto de directivas de los bucles paralelos, la directiva `nc_result` no se sitúa entre el constructo y el bucle `for`, sino en el interior del bloque de código del bucle, inmediatamente antes del acceso no contiguo a memoria.

```
1  int v[N];

3  #pragma omp parallel for
4  for (i = 0; i < N; i++) {
5     #pragma llc nc_result (&v[f(i)], 1, &v[0])
6     v[f(i)] = g(i);
7  }
```

Listado 2.4: Directiva `nc_result` en accesos no contiguos a memoria

El ejemplo del listado 2.4 muestra una posible situación donde se realiza un acceso al vector `v` con patrón no conocido.

En cada iteración `i` se accede al elemento `f(i)` del vector `v` (línea 6), no pudiendo determinarse esta posición hasta el instante que se ejecuta dicha línea. Estas situaciones se resuelven en `llc` mediante la directiva `nc_result` que se inserta en la línea 5 y que será evaluada por el compilador en tiempo de ejecución justo antes de realizar el acceso a memoria.

La directiva `nc_result` lleva implícita una sincronización de todos los procesadores del grupo al finalizar la ejecución del bucle paralelo.

2.5.2.4. Directiva `lastresult`

Las variables que son actualizadas dentro de un bucle conservan como valor final el obtenido en la asignación de la última iteración. Este hecho que es obvio en el caso secuencial, no tiene por qué serlo necesariamente en las ejecuciones en paralelo, donde distintos grupos de procesadores ejecutan distintas iteraciones en un orden no predefinido.

En el caso paralelo, al finalizar la ejecución del bucle, cada subgrupo de procesadores tendrá como valor final el correspondiente a la última iteración

que llevó a cabo, por lo que sólo el subgrupo que ejecutó la última iteración del bucle dispondrá del valor correcto. Si este valor es usado una vez finalizado el bucle, debe asegurarse que en todos los subgrupos coincide con el de la última iteración.

Para esta situación, `llc` provee una directiva que indica a los procesadores que ejecutaron la última iteración la necesidad de comunicar el valor de las variables especificadas a los demás procesadores de los restantes subgrupos. La sintaxis de esta directiva es la siguiente:

```
#pragma llc lastresult (p, n, {, p, n,}*)
```

1. `p`: puntero al principio de la zona de memoria a comunicar.
2. `n`: número de elementos a comunicar a partir de la dirección indicada por `p`.

Existe una cláusula en `OpenMP` que tiene este mismo efecto. `llc` permite que el usuario indique qué variables deben ser comunicadas mediante el uso de una de estas alternativas, o bien la directiva específica de `llc` mostrada, o bien a través de la cláusula de `OpenMP` que a continuación se presenta:

```
#pragma ... lastprivate (var {, var}*)
```

donde `var` indica la variable o lista de variables que deben ser comunicadas.

El ejemplo mostrado en el listado 2.5 expone una situación en la que debe usarse la directiva `lastresult`. Los valores de la variable `a` y el vector `v` son usados por la función `h` (línea 15) al finalizar el bucle. Por este motivo, el grupo de procesadores que realice la última ejecución debe comunicar los valores de estas variables a todos los restantes grupos. Este comportamiento se consigue mediante las directivas descritas.

En este listado 2.5 hemos mostrado el uso conjunto de la cláusula de `OpenMP lastprivate` para la variable `a` y la directiva de `llc lastresult` para el vector `v`. Para los vectores se aconseja el uso de las directivas de `llc`, ya que permiten especificar qué porciones del vector se deben transmitir.

```
1   int v[M];
2   int a;

4   init(v);
5   #pragma omp parallel for lastprivate (a)
6   #pragma llc lastresult (&v[0], M)
7    for (i = 0; i < N; i++) {
8      for (j = 0; j < M; j++) {
9        v[j] = f(i, j);
10     }
11     ...
12     a = g(i);
13     ...
14   }
15   h (v, a);
```

Listado 2.5: Directiva `lastresult`

Como se ha comentado anteriormente, el usuario puede escoger entre utilizar sólo cláusulas de OpenMP, sólo directivas de `llc` o una combinación de ambas, dependiendo de cada código en particular. El código del listado 2.6 muestra una situación donde el mismo código es válido tanto para OpenMP como para `llc` sin tener que realizar modificación alguna sobre el programa OpenMP original.

```
1   int a;

3   init(v);
4   #pragma omp parallel for lastprivate (a)
5    for (i = 0; i < N; i++) {
6      a = g(i)
7    }
8    h (a);
```

Listado 2.6: Cláusula `lastprivate`

La directiva `last_result` lleva implícita una sincronización de todos los procesadores del grupo al finalizar la ejecución del bucle paralelo.

2.5.2.5. Directiva `reduce`

La directiva `reduce` permite aplicar una *operación de reducción* sobre una variable al realizar la comunicación. Estas operaciones de reducción se aplican cuando el valor final de una variable se calcula aplicando una operación definida sobre los resultados parciales obtenidos en paralelo.

En los bucles `for` se debe usar esta directiva cuando en cada iteración se calcula una parte de la solución final, siendo necesario combinar los resultados parciales para obtener el valor final. La directiva de reducción en `llc` tiene la siguiente sintaxis:

```
#pragma llc reduce (p_d, p_s, n, op {, p_d, p_s, n, op}*)
```

1. `p_d`: puntero a la variable que contiene las contribuciones parciales.
2. `p_s`: puntero a la variable que en la que se almacenará la solución final. Esta variable debe ser diferente de `p_d`. *Todos* los procesadores del grupo deben indicar su contribución mediante el puntero `p_d` y *todos* ellos recibirán el resultado final en la variable apuntada por `p_s`.
3. `n`: `llc` permite realizar operaciones de reducción sobre variables simples o bien total o parcialmente sobre vectores, llevando a cabo la operación elemento a elemento. Mediante el parámetro `n` el usuario indica el número de elementos sobre los que se efectuará la operación, empezando en la dirección indicada por los punteros.
4. `op`: Este parámetro especifica qué operación de reducción se aplicará. Esta operación puede ser una de las predeterminadas de `llc` mostradas en la tabla 2.1 o bien una definida por el usuario.

Las operaciones de reducción han de cumplir ciertas condiciones:

- La operación debe cumplir la propiedad *asociativa* y *conmutativa* ya que el orden en el que se combinen las soluciones parciales no debe repercutir en la solución final.
- Las variables afectadas por la operación de reducción deben estar inicializadas al *elemento neutro* de la operación. Si no es así, no se puede asegurar que el resultado obtenido sea el correcto.

La tabla 2.1 recoge las operaciones de reducción predefinidas en `llc`, indicando para cada operación el elemento neutro al que deben estar inicializadas las variables.

ID Oper.	**Descripción**	Neutro
`LLC_SUM`	Suma	0
`LLC_PROD`	Producto	1
`LLC_SUBS`	Resta	0
`LLC_MAX`	Máximo	- ∞
`LLC_MIN`	Mínimo	+ ∞
`LLC_LOR`	*OR* lógico	0
`LLC_LAND`	*AND* lógico	1
`LLC_LXOR`	*XOR* lógico	0
`LLC_BOR`	*OR* bit a bit	0
`LLC_BAND`	*AND* bit a bit	˜0
`LLC_BXOR`	*XOR* bit a bit	0

Tabla 2.1: Operaciones de reducción predefinidas en `llc`.

Además de estas operaciones predefinidas, `llc` permite el uso de *funciones definidas por el usuario*. La sintaxis de estas funciones debe ser la siguiente:

```
t0 nombre_funcion (t1 *ptr_data, t1 *ptr_sol, t2 num_elem)
```

La función debe cumplir con las siguientes restricciones:

- *`nombre_funcion`* es el identificador que el usuario deberá especificar como argumento `op` en la directiva `reduce`.
- `t1` corresponde con cualquier tipo de datos válido que sea compatible con la operación de reducción implementada. La función debe recibir dos punteros a variables de tipo `t1`, donde el primer puntero (`ptr_data`) apunta a la variable que contiene las contribuciones parciales y el segundo puntero (`ptr_sol`) debe apuntar a la variable en la que se almacenará la solución final.
- `t2` es un tipo de datos entero (`int`, `long`, etc.) y el argumento `num_elem` indica sobre cuántos elementos se aplicará la operación de reducción.
- `t0` es cualquier tipo válido y `llc` no pone restricciones sobre si la función debe o no devolver un valor y cuál es su tipo.

- El usuario es el responsable de implementar una función que cumpla las propiedades que deben tener las operaciones de reducción.

`OpenMP` dispone de una cláusula para reducciones en los bucles paralelos. `llc` reconoce y permite el uso de esta cláusula, cuya sintaxis es la siguiente:

```
#pragma omp [parallel] for ... reduction (op: var {, var}*) ...
```

donde `op` es la operación de reducción que va seguida por una lista de una o más variables `var` sobre las que se aplicará la operación `op`. En el estándar 2.5 de `OpenMP` se permiten las operaciones de reducción mostradas en tabla 2.2.

Op.	Descripción	Neutro
+	Suma	0
*	Producto	1
-	Resta	0
|	*OR* bit a bit	0
&	*AND* bit a bit	~0
^	*XOR* bit a bit	0
||	*OR* lógico	0
&&	*AND* lógico	1

Tabla 2.2: Operaciones de reducción permitidas por `OpenMP`.

`llc` es compatible con la cláusula `reduction` de `OpenMP` y si el usuario especifica una o más de estas cláusulas, `llc` realizará las operaciones de reducción indicadas. Sin embargo, para realizar estas operaciones, el modelo implementado en `llc` necesita conocer el tipo de las variables afectadas[3]. Esta directiva de `llc` sólo debe usarse en combinación de la cláusula `reduction` de `OpenMP`, colocándose inmediatamente a continuación del constructo del bucle paralelo, antes de cualquier otra directiva de `llc`. La sintaxis de esta directiva es:

```
#pragma llc reduction_type (type {, type}*)
```

[3]Véase el apartado 3.6.5 (página 143) que estudia la implementación de este constructo.

donde el argumento de esta directiva es un listado de los tipos de datos que corresponden a las variables que aparecen en la cláusula o cláusulas `reduction` de `OpenMP`, respetándose estrictamente el orden de aparición.

El uso de la directiva de `llc` para las reducciones presenta una serie de ventajas frente a la cláusula de `OpenMP`. Una de ellas es la posibilidad que se le ofrece al usuario de poder definir sus propias operaciones de reducción. Por otro lado, `llc` elimina algunas de las restricciones de `OpenMP` en cuanto al tipo de variables. Mientras `llc` permite realizar operaciones de reducción elemento a elemento de forma total o parcial sobre un vector, `OpenMP` no permite el uso de tipos derivados (incluyendo vectores), punteros ni referencias.

El listado 2.7 presenta un código que recoge varias de las situaciones comentadas en este apartado: el vector `v` es reducido elemento a elemento mediante una operación `OR` bit a bit (directiva `reduce` de `llc`, línea 17). Por otro lado, tanto el tipo de la variable `r` como la operación de reducción que se le aplicarla (`suma_real`) han sido definidas por el usuario (listado 2.8, página 74) y son usadas en la directiva `reduce` en la línea 17.

Además de estas operaciones específicas de `llc`, el listado 2.7 también recoge reducciones que sí pueden ser indicadas mediante la cláusula `reduction` de `OpenMP` (línea 15), que se complementa con la directiva `reduction_type` de `llc` en la línea 16.

La directiva `reduce` lleva implícita una sincronización de todos los procesadores del grupo al finalizar la ejecución del bucle paralelo.

2.5.2.6. Directiva `weight`

La función de esta directiva es la de proporcionar al compilador la información necesaria para efectuar un equilibrado de carga al realizar el mapeado entre los grupos procesadores y las tareas que le serán asignadas. Si no se especifica información de pesos, el compilador realizará una asignación equitativa, tratando de adjudicar igual número de tareas a cada grupo de procesadores.

Sin embargo, si el usuario dispone de información sobre el *peso computacional* de cada tarea, el compilador tratará de usar esta información para realizar una ejecución más equilibrada y asignará más recursos a aquellas tareas más pesadas. Este peso no tiene por qué ser un valor exacto ni absoluto, el usuario puede realizar una estimación y dar un peso relativo de unas tareas frente a otras. La sintaxis de esta directiva es:

```
1   real r, r_sol;
2   int a;
3   float b;
4   long v[N], v_sol[N];

6   /* Inicialización a elementos neutros */
7   r.re = 0.0;       r.im = 0.0;
8   r_sol.re = 0.0;   r_sol.im = 0.0;
9   a = 1;            b = 1.0;
10  for (i = 0; i < N; i++) {
11    v[i] = 0;
12    v_sol[i] = 0;
13  }
14  ...
15  #pragma omp parallel for reduction (*: a, b)
16  #pragma llc reduction_type (int, float)
17  #pragma llc reduce (&v[0], &v_sol[0], N, LLC_BOR, \
18                      &r, &r_sol, 1, suma_real);
19  for (i = 0; i < LIM; i++) {
20    a *= f(i);
21    b *= g(i);
22    r.re += h_re(i);
23    r.im += h_im(i);
24    for (j = 0; j < N; j++) {
25      v[j] |= l(i, j);
26    }
27  }
```

Listado 2.7: Reducciones en bucles paralelos

```
2   typedef struct {
3     double re;
4     double im;
5   } real;

7   void suma_real (real *data, real *sol, int n) {
8     int i;

10    for (i = 0; i < n; i++) {
11      sol[i].re += data[i].re;
12      sol[i].im += data[i].im;
13    }
14  }
```

Listado 2.8: Definiciones de tipo y función para la reducción

```
#pragma llc weight (w)
```

donde `w` indica el peso y debería tener un valor que dependa de cada tarea. La expresión `w` es evaluada en tiempo de ejecución y el valor de esta expresión para cada iteración debe ser un número entero no negativo. Esta directiva debe estar situada inmediatamente después del constructo del bucle paralelo (con posterioridad a la directiva `reduction_type` si se hubiese especificado).

```
1   int v[N];

3   init(v);
4   #pragma omp parallel for
5   #pragma llc weight (fin(i) - ini(i) + 1)
6   for (i = 0; i < N; i++) {
7     f(v, ini(i), fin(i));
8   }
```

Listado 2.9: Directiva `weight` de pesos de las tareas

El ejemplo[4] del listado 2.9 ilustra una situación donde se ha aplicado la directiva de equilibrado. Consiste en la paralelización de un bucle en el que

[4]El ejemplo sólo tiene aplicación didáctica, ya que no se ha utilizado ninguna directiva de comunicación de resultados.

en cada iteración `i` se aplica una función `f` (línea 7) sobre el vector `v` que afecta a las posiciones comprendidas entre los valores devueltos por `ini(i)` y `fin(i)`.

Si el usuario no especifica los pesos, el compilador generará el código que tratará de realizar una asignación equitativa de procesadores a cada tarea. Esta asignación puede ocasionar un rendimiento pobre, ya que unas porciones de vectores serán más grandes que otras y el compilador está asignando igual número de recursos a todas ellas.

Si el usuario especifica el peso estimado para cada tarea, que en este caso consiste en el número de elementos sobre los que actuará la función en cada iteración (`fin(i) - ini(i) + 1`), el compilador tendrá en cuenta esta información para realizar una distribución proporcional a cada peso indicado, asignando más recursos a las tareas con mayor peso.

Esta directiva tiene un especial interés en paralelismo multinivel (mediante recursividad o anidamiento), ya que creará grupos de procesadores más grandes para las tareas más pesadas, permitiendo que la profundidad del paralelismo anidado sea mayor donde se localicen estas tareas con más peso.

2.5.3. Resumen de la sintaxis del constructo `for`

A continuación se presenta un resumen de la sintaxis `llc` para los bucles `for` paralelos donde se indican los argumentos de cada directiva y cómo deben usarse:

```
 #pragma omp [parallel(a)] for {omp_for_clause(b)}*
[#pragma llc reduction_type(c) (type {, type}*)]
[#pragma llc weight (w)]
{#pragma llc result (p, n {, p, n}*) |
 #pragma llc rnc_result (p, n, m, r {, p, n, m, r}*) |
 #pragma llc lastresult (p, n {, p, n}*) |
 #pragma llc reduce (p_data, p_sol, n, op {, p_data, p_sol, n, op}*)
}*
   for (...)
    {
      ...
      {#pragma llc nc_result(p, n, v {, p, n, v}*)}*
      ...
    }
```

[a]: Si la palabra clave `parallel` no se especifica, el constructo debe estar dentro de una región paralela.
[b]: Todas las cláusulas `OpenMP` son admitidas por razones de compatibilidad. Sin embargo, sólo tendrán efecto en `llc` las cláusulas `reduction` y `lastprivate`. Para el resto de casos donde es necesario modificar los atributos de las variables compartidas dentro de los bucles paralelos, el usuario deberá usar las correspondientes directivas `llc` en lugar de las cláusulas de `OpenMP`.
[c]: La directiva de `llc` `#pragma llc reduction_type` sólo se usa para completar la cláusula `reduction` de `OpenMP`, en caso de que sea usada.

2.6. Constructo `sections`

Este método de paralelización consiste en la división de un código secuencial en diferentes bloques, de forma que cada bloque sea independiente del resto y pueda ejecutarse en paralelo.

Los constructos `sections` y `section` dan soporte a este tipo de paralelismo. La primera de estas directivas (`sections`) indica el bloque de código que engloba a todas las secciones que se van a ejecutar en paralelo, mientras que con el constructo `section` marcamos los bloques de código que forman cada sección paralela.

2.6.1. Ámbito de aplicación del constructo `sections` en `llc`

El constructo `sections` de `llc` puede aplicarse en las mismas situaciones que el equivalente de `OpenMP`. En general, cualquier bloque de código estructurado puede ser susceptible de ser paralelizado mediante la partición del mismo en secciones paralelas, siempre que se cumplan las siguientes restricciones:

- Cada sección debe contener un bloque estructurado de código independiente del resto.
- El orden de ejecución de las secciones no debe ser relevante.
- Cualquier código dentro del bloque indicado por el constructo `sections` debe pertenecer a una y solamente a una sección .

La sintaxis del constructo `sections` es la siguiente:

```
#pragma omp [parallel] sections {omp_sections_clauses}*
```

Dentro de la región definida por el constructo `sections` pueden aparecer una o más instancias del constructo `section`, cuya sintaxis se expresa a continuación:

```
#pragma omp section
```

En `OpenMP` el uso del constructo `section` es opcional para la primera sección. Sin embargo, en `llc` el constructo `section` debe usarse para marcar todas las regiones.

2.6.2. Directivas `llc` del constructo `sections`

Las diferentes directivas que el constructo `sections` permite ya han sido definidas en el apartado 2.5 dedicado al constructo `for`. Las consideraciones sobre las directivas que se han realizado en dicho apartado se aplican también en éste, teniendo en cuenta el cambio de contexto de *bucle paralelo* por el de *secciones paralelas* e *iteraciones* por *secciones*. La explicación se cada directiva tomará como base el ejemplo del listado 2.10.

2.6.2.1. Directiva `result`

La directiva `result` se utiliza en el ámbito del constructo `section` para identificar qué zonas de memoria han sido accedidas para escritura en cada sección. Estas regiones deben ser comunicadas a los otros subgrupos de procesadores para asegurar la consistencia de memoria según el modelo OTOSP.

En el ejemplo del listado 2.10 cada sección comunica al resto de subgrupos los resultados obtenidos tras aplicar la función `do_complex_operations()` sobre el correspondiente vector. Para ello utiliza las directivas `result` (líneas 14 y 25), identificando las regiones de memoria por un puntero al principio de estas zonas y el número de elementos afectados.

Para más detalles, significado, modo de uso y sintaxis de esta directiva, consultar el apartado 2.5.2.1 (página 63).

```
1   b = 0;
2   c = 1;
3   c_sol = 1;
4   ...
5   #pragma omp parallel sections  \
6            lastprivate (a) reduction (+: b)
7   #pragma llc reduction_type (int)
8   #pragma llc reduce (&c, &c_sol, 1, LLC_PROD)
9   #pragma llc lastresult (&d, 1)
10  {

12    #pragma omp section
13    #pragma llc weight((N * N))
14    #pragma llc result (&v1[0], N)
15    {
16      do_complex_operations (v1, N);
17      a = g1();
18      b += h1();
19      c *= l1();
20    }  /* Sección 1 */


23    #pragma omp section
24    #pragma llc weight((M * M))
25    #pragma llc result (&v2[0], M)
26    {
27      do_complex_operations (v2, M);
28      a = g2();
29      b += h2();
30      c *= l2();
31    }  /* Sección 2 */

33  } /* Constructo sections */
```

Listado 2.10: Secciones paralelas

2.6.2.2. Directiva `reduce`

El constructo `sections` admite la directiva `reduce` para indicar que a una o más variables se les debe aplicar una operación de reducción. Luego, el valor final resultante tras la aplicación de la operación, deberá ser comunicado a todos los procesadores del grupo.

Las variables afectadas en la reducción deben ser inicializadas al valor neutro de la operación antes de entrar en la región delimitada por el constructo `sections`. En caso de no llevar a cabo esta inicialización o de realizarse dentro del ámbito del constructo, no puede asegurarse que el resultado obtenido sea correcto.

Como en el caso del constructo `for`, se permite que el usuario utilice la cláusula `reduction` de `OpenMP` para especificar la operación de reducción, debiendo en ese caso completar la información mediante la directiva de `llc reduction_type`.

En el ejemplo que se presenta en el listado 2.10, aplicamos operaciones de reducción sobre las variables `b` y `c`. La primera de ellas (variable `b`) consiste en una operación de suma que se especifica mediante la cláusula de `OpenMP reduction` (línea 6) y se complementa con la directiva de `llc reduction_type` (línea 7). A la segunda variable (`c`) se le aplica una operación de producto que se indica a través de la directiva `llc reduce` de la línea 8 y cuyo resultado final se obtiene en la variable `c_sol`. Como queda reflejado en este ejemplo, todas las variables afectadas por las operaciones de reducción (`b`, `c` y `c_sol`) son inicializadas a los respectivos elementos neutros de cada operación, llevándose a cabo esta inicialización antes de entrar a la región de las secciones (líneas 1 a 3).

Para más detalles, significado, modo de uso y sintaxis de estas directivas, consultar el apartado 2.5.2.5 (página 69).

2.6.2.3. Directiva `weight`

Esta directiva puede especificarse junto con el constructo `section` para indicar el *peso* relativo o estimación del coste computacional de una sección con respecto a las demás.

El compilador de `llc` se basa en esta información para la asignación de recursos a cada sección, con el fin de proporcionar un correcto equilibrado de cargas. De este modo, se asignarán más recursos a aquellas secciones con una mayor carga computacional.

En el ejemplo del listado 2.10 se ha determinado que el coste computacional de cada sección se puede estimar directamente a partir de la función `do_complex_operations` y este coste depende cuadráticamente de la cantidad de elementos del vector sobre los que se realizará la operación. De este modo, la estimación de pesos es de N^2 para la primera sección (línea 13) y M^2 para la segunda (línea 24). El compilador de `llc` realizará una distribución de recursos proporcional a estos valores.

Se debe tener en cuenta que los valores de los pesos indicados en la `weight` deben ser enteros positivos. En caso de que esta directiva `weight` no sea utilizada en todas las secciones, se asume que aquellas secciones sin esta directiva tienen un peso por defecto igual a `1`.

Para más detalles, significado, modo de uso y sintaxis de esta directiva, consultar el apartado 2.5.2.6 (página 72).

2.6.2.4. Directiva `lastresult`

Esta directiva pertenece al ámbito del constructo `sections`. Su significado coincide con el de la misma directiva de los bucles paralelos, pero en este caso los valores a comunicar corresponden con los obtenidos en la ejecución de la última sección. Nótese que nos referimos a la ejecución de la sección que aparece en último lugar en el código, lo que no tiene por qué coincidir necesariamente con la última sección ejecutada en paralelo. `llc` admite el uso alternativo de la cláusula `lastprivate` de `OpenMP` para este mismo fin.

En el ejemplo que se muestra en el listado 2.10 se pueden localizar dos variables afectadas por estas directivas. Por un lado la variable `a` que indica la cláusula `lastprivate` de `OpenMP` (línea 6) y, por otro, la variable `d` de la directiva de `llc lastresult` (línea 9). En este ejemplo, tras finalizar la ejecución de todas las secciones, ambas variables tendrán los valores asignados durante la ejecución de la sección 2.

Para más detalles, significado, modo de uso y sintaxis de esta directiva, consultar el apartado 2.5.2.4 (página 66).

2.6.3. Resumen de la sintaxis del constructo `sections`

`#pragma omp` [`parallel`$^{(a)}$] `sections` {*omp_sections_clause*$^{(b)}$}*
[`#pragma llc reduction_type`$^{(c)}$ (*type* {, *type*}*)]
{`#pragma llc lastresult` (*p, n* {, *p, n*}*) |
`#pragma llc reduce` (*p_data, p_sol, n, op* {, *p_data, p_sol, n, op*}*)

}*
sections_code

Las secciones deben especificarse dentro del bloque *sections_code*. Todo código dentro de esta región debe pertenecer a una y sólo a una sección que debe ser declarada usando el constructo `section` usando para ello la siguiente sintaxis:

```
#pragma omp section
[#pragma llc weight (w)]
{#pragma llc result (p, n {, p, n}*)}*
```
section_code

(a): Si la palabra clave `parallel` no se especifica, el constructo debe estar dentro de una región paralela.
(b): Todas las cláusulas `OpenMP` son admitidas por razones de compatibilidad. Sin embargo, sólo tendrán efecto en `llc` las cláusulas `reduction` y `lastprivate`. Para el resto de casos donde es necesario modificar los atributos de las variables compartidas dentro de las secciones paralelas, el usuario deberá usar las correspondientes directivas `llc` en vez de las cláusulas de `OpenMP`.
(c): La directiva de `llc #pragma llc reduction_type` sólo es utilizada para completar la cláusula de `OpenMP reduction`, en caso de que sea usada.

2.7. Constructo `pipeline`

2.7.1. Introducción

El paradigma de paralelismo *pipeline* o segmentado se aplica cuando en una aplicación el procesado de los datos se lleva a cabo en series o etapas, de forma que la salida de una de estas etapas es la entrada de la siguiente y así sucesivamente. Si estas etapas cumplen con las restricciones impuestas, entonces podrán ser ejecutadas en paralelo, teniendo siempre en cuenta las dependencias que cada etapa tiene con las etapas anteriores y/o posteriores [131].

Un ejemplo muy usual de este tipo de paralelismo se encuentra en aquellas aplicaciones que requieren el procesado de matrices. Normalmente, la actualización de un elemento de la matriz depende de los datos que se han calculado previamente para uno o varios elementos contiguos. De este modo, la actualización de uno (o varios) de estos elementos puede ser considerada

como una etapa y la paralelización de la aplicación puede llevarse a cabo mediante un paralelismo de segmentación.

2.7.2. Ámbito de aplicación del constructo `pipeline` en `llc`

`OpenMP` en su estándar 2.5 no da soporte al paralelismo de segmentación, por lo que los constructos que se presentamos en esta sección son específicos de `llc`.

El soporte a los pipelines que `llc` ofrece se basa en bucles de tipo `for` donde cada iteración o grupo de ellas puede ser considerada como una etapa. Esta aproximación permite la paralelización de aplicaciones que contengan bucles que recorren etapas, como el ejemplo indicado de procesado de matrices. De este modo, el usuario deberá construir el bucle `for` de modo que en cada iteración i se actualicen aquellos elementos de la matriz que dependen de los resultados obtenidos en la iteración previa $i-1$ y, que a su vez, los datos que se actualicen en la iteración i sirvan como entrada para los cálculos realizados en la iteración posterior $i+1$.

El constructo de `llc` que define la paralelización de pipelines es el siguiente:

```
#pragma llc pipeline
```

2.7.3. Directivas `llc` del constructo `pipeline`

El constructo `pipeline` se sirve de tres directivas para marcar qué variables son compartidas y deben comunicarse. Dos de estas directivas son exclusivas de los pipelines y posibilitan la comunicación de datos entre etapas consecutivas, por lo que una etapa puede recibir datos de la fase anterior y/o enviar los resultados a la etapa posterior. La tercera directiva es general y su función es la de permitir que los resultados finales sean distribuidos entre todos los procesadores del grupo original, asegurando así la consistencia de memoria según el modelo OTOSP.

2.7.3.1. Directiva `send`

Esta directiva permite enviar datos de una etapa a la etapa posterior, siempre y cuando no sea trate de la última etapa. La directiva `send` de `llc` lleva implícita la comprobación de si está ejecutando la última etapa, caso en el que no se realiza la comunicación.

La directiva `send` se debe utilizar directamente en el bloque de código del bucle `for`, justo en la posición en la que se desee realizar el envío de datos. La sintaxis de esta directiva es la siguiente:

```
#pragma llc send (p, n {, p, n}*)
```

1. `p`: puntero al principio de la zona de memoria a comunicar.
2. `n`: número de elementos a comunicar a partir de la dirección indicada por `p`.

2.7.3.2. Directiva `receive`

La directiva `receive` de `llc` tiene la funcionalidad complementaria a la directiva `send`. En este caso, se reciben los datos de la etapa previa, siempre que no se esté en la etapa inicial. Como en el caso anterior, la comprobación de etapa inicial es realizada implícitamente en el código generado.

Al igual que la directiva `send`, la directiva `receive` debe ser utilizada directamente en el bloque de código del bucle `for`, situándola en aquella posición en la que se desee realizar la recepción de datos de la etapa previa. La sintaxis de la directiva en cuestión es la siguiente:

```
#pragma llc receive (p, n {, p, n}*)
```

1. `p`: puntero al principio de la zona de memoria a comunicar.
2. `n`: número de elementos a comunicar a partir de la dirección indicada por `p`.

2.7.3.3. Directiva `result`

El uso de esta directiva en los pipelines permite que los subgrupos de procesadores que han ejecutado cada etapa o conjunto de etapas, comuniquen al resto de subgrupos los resultados obtenidos, de forma que se asegure la coherencia de memoria según el modelo OTOSP y pueda realizarse la unión al grupo inicial de procesadores al finalizar el pipeline.

Para más detalles, significado, modo de uso y sintaxis de esta directiva, consultar el apartado 2.5.2.1 (página 63).

2.7.4. Resumen de la sintaxis del constructo `pipeline`

A continuación se expone un resumen de la sintaxis de las directivas del constructo `pipeline`.

```
#pragma llc [parallel] pipeline
{#pragma llc result (p, n {, p, n}*)}*

  for (...) {
   ...
   {#pragma llc send (p, n {, p, n}*) |
   #pragma llc receive (p, n {, p, n}*)}*
   ...

  }
```

2.8. Constructo `taskq`

El modelo `workqueuing` o paralelismo de colas de tareas permite la paralelización de aplicaciones que incluyen estructuras de datos dinámicas con patrones irregulares o aplicaciones con complejas secuencias de control. Como ejemplos concretos podemos citar los bucles que dependen de una condición de parada (como bucles `while` o aquellos que incluyen sentencias que rompen el flujo de control, como `break` o `continue`, `goto`, ...), estructuras de datos complejas, como listas de punteros, árboles, etc.

Mediante el modelo de colas de tareas, cada vez que se encuentra una tarea paralela en tiempo de ejecución, esta tarea es colocada en una cola de tareas por ejecutar. Cuando un proceso o thread esté disponible, extraerá una

tarea de la cola y procederá a ejecutarla. Este procedimiento se repite tantas veces como sea necesario hasta que la cola de tareas se vacíe.

2.8.1. Ámbito de aplicación del constructo taskq en llc

El grupo de KAI-intel [145] ha sido uno de los primeros en incluir en un compilador de OpenMP de uso general los constructos que dan soporte a este modelo. La forma de realizar la paralelización propuesta por el grupo de KAI-Intel se realiza mediante el uso de dos directivas que son utilizadas de un modo que guarda cierta similitud al explicado para el constructo sections.

El primero de estos constructos se denomina taskq y es usado para delimitar el bloque de código en cuyo interior pueden aparecer las tareas paralelas. La sintaxis de esta directiva es la que sigue:

```
#pragma intel omp [parallel] {omp_taskq_clauses}*
```

Para marcar estas tareas paralelas, en el interior de una región taskq se debe usar una o varias instancias del segundo constructo: task. Este constructo engloba el bloque de código que constituye la tarea que será ejecutada en paralelo y su sintaxis es la siguiente:

```
#pragma intel omp {omp_task_clauses}*
```

Las siguientes restricciones se imponen para una aplicación llc válida en este modelo:

- En bucles con condición de parada (como por ejemplo bucles while), la evaluación de esta condición no puede depender directamente de cálculos realizados dentro de las tareas paralelas.
- Se permite el uso sentencias de continuación o parada en bucles (como continue, break, etc.) dentro de la región especificada por el constructo taskq, siempre que estas sentencias se encuentren fuera de los bloques de código de las tareas paralelas indicados por medio del constructo task.

- En su versión actual, `llc` no permite paralelismo recursivo en este constructo ni el anidamiento de tareas.

El grupo de `KAI-Intel` propuso este modelo [139] para ser incorporado en el nuevo estándar de `OpenMP` 3.0 [124]. En el momento de inicio de nuestro desarrollo, decidimos adoptar la sintaxis propuesta por `KAI-Intel` [60] porque estaba implementada en el compilador `icc` de `Intel` y era prácticamente la única alternativa disponible en aquellas fechas (2006).

El estándar de `OpenMP` 3.0 fue finalmente publicado en mayo de 2008, tras un largo periodo de revisión y discusión. La sintaxis final empleada en este estándar difiere a la propuesta por `KAI-Intel`, aunque en esencia conserva sus principios. La directiva `taskq` no aparece en el estándar, si bien sí se preserva el uso de la directiva `task`, que denota el bloque de código de la tarea dentro de una región paralela, y que incluye algunas cláusulas específicas, como `untied`. El estándar también incorpora una directiva `taskwait` que espera a que se completen todas las tareas hijas generadas desde el principio de la tarea actual.

El enfoque de `OpenMP` 3.0 aporta la ventaja de permitir el anidamiento y recursividad de tareas, para lo cual se han definido dos nuevas variables de entorno que controlan el nivel activo de anidamiento actual y el máximo de threads que participan en el programa `OpenMP`, así como un cierto número de funciones que permiten obtener y modificar el valor de estas variables durante la ejecución.

No obstante, la implementación de las tareas de `OpenMP` también ha recibido algunas críticas [144], como la dificultad de obtener una implementación eficiente de ciertos algoritmos con la nueva API. Por ejemplo, la eficiencia de las aplicaciones que trabajan con listas enlazadas sigue dependiendo del sacrificio en escalabilidad y rendimiento o bien de la necesidad de añadir código específico de `OpenMP`, como llamadas a las funciones anteriormente mencionadas, lo que a nuestro criterio rompe con la filosofía de mantener la funcionalidad del código secuencial original.

2.8.2. Directivas `llc` de los constructo `taskq` y `task`

`llc` propone el uso de sus propias directivas frente a la utilización de las cláusulas no estándar propuestas por el grupo `KAI-Intel`. Como se estudiará en el capítulo 3, el modelo *workqueuing* se ha implementado en `llc` mediante un enfoque *maestro-esclavo*.

llc sigue el convenio de utilizar el término master en el nombre de las directivas que involucran directamente al maestro, y el término slave para las directivas de los esclavos. Sin embargo, el usuario puede obviar esta diferenciación ya que la implementación que utiliza llc para dar soporte al modelo es transparente al programador. El papel del usuario se centra en determinar qué datos iniciales deben ser conocidos por la tarea para poder ejecutarse y qué resultados parciales produce la tarea tras su ejecución y deben ser compartidos, lo que equivale a comunicarlos. Otro convenio seguido en llc es utilizar el prefijo "taskq_" para las directivas del constructo taskq, mientras que el prefijo "task_" pertenece a las directivas del constructo task.

2.8.2.1. Directiva taskq_master_result

Ésta es la única directiva llc que acepta el constructo taskq. Su uso permite asegurar la consistencia de memoria como requiere el modelo OTOSP. Tras la ejecución de una aplicación en este modelo, solamente el procesador maestro dispondrá de todos los resultados finales. En este caso, el usuario debe utilizar una o varias instancias de esta directiva para indicar la localización en memoria de estos resultados con el fin de que el maestro los distribuya al resto de procesadores en el grupo, logrando así que todos tengan la misma visión de memoria y puedan reunirse en el grupo original.

La sintaxis de esta directiva es la siguiente:

```
#pragma llc taskq_master_result (p, n {, p, n}*)
```

1. p: puntero al principio de la zona de memoria a comunicar.
2. n: número de elementos a comunicar a partir de la zona de memoria indicada por p.

Esta directiva presenta unas características y funcionalidades similares a la directiva result presentada en el apartado 2.5.2.1 (página 63). Para más detalles, puede consultarse esta referencia, teniendo en cuenta las diferencias obvias entre los modelos de bucles for paralelos y de colas de tareas.

2.8.2.2. Directiva task_master_data

La directiva task_master_data se usa en el constructo task para indicar qué datos debe obtener inicialmente una tarea para hacer posible la ejecución,

lo que equivale en el modelo *maestro-esclavo* a los datos que el maestro deberá enviar al esclavo encargado de llevar a cabo la ejecución. La sintaxis de esta directiva es la que a continuación se presenta:

```
#pragma llc task_master_data(p, n {, p, n}*)
```

1. `p`: puntero al principio de la zona de memoria a comunicar.
2. `n`: número de elementos a comunicar a partir de la zona de memoria indicada por `p`.

Esta directiva presenta unas características y funcionalidades similares a la directiva `result` explicada en el apartado 2.5.2.1 (página 63). Para más detalles, puede consultarse esta referencia, teniendo en cuenta las diferencias obvias entre los modelos de bucles `for` paralelos y el de colas de tareas.

2.8.2.3. Directiva `task_slave_data`

Esta directiva es se usa en el constructo `task` para indicar los resultados parciales que han sido obtenidos tras la ejecución de la tarea por parte del esclavo y que deben ser comunicados al maestro. La sintaxis de esta directiva se muestra a continuación:

```
#pragma llc task_slave_data(p, n {, p, n}*)
```

1. `p`: puntero al principio de la zona de memoria a comunicar.
2. `n`: número de elementos a comunicar a partir de la zona de memoria indicada por `p`.

Esta directiva presenta unas características y funcionalidades similares a la directiva `result` que fue presentada en el apartado 2.5.2.1 (página 63). Para más detalles, puede consultarse esta referencia, teniendo en cuenta las diferencias obvias entre los modelos de bucles `for` paralelos y de colas de tareas.

2.8.2.4. Directiva task_slave_rnc_result

La directiva task_slave_rnc_result es usada en el ámbito del constructo task con el mismo fin que la anterior directiva task_slave_data. La diferencia entre ambas es que la directiva task_slave_rnc_result se utiliza para indicar regiones de memoria no contiguas, pero que siguen un patrón de comunicaciones regular, siendo su sintaxis la siguiente:

```
#pragma llc rnc_result (p, n, m, r {, p, n, m, r}*)
```

1. p: puntero al principio de la zona de memoria a comunicar.
2. n: número de elementos a comunicar a partir de la dirección especificada por p.
3. n: número de elementos que no serán comunicados a partir de los m comunicados.
4. r: número de veces que se repetirá el patrón de comunicación descrito.

Esta directiva presenta unas características y funcionalidades similares a la directiva rnc_result presentada en el apartado 2.5.2.2 (página 64). Para más detalles, puede consultarse esta referencia, teniendo en cuenta las diferencias obvias entre los modelos de bucles for paralelos y de colas de tareas.

2.8.2.5. Directiva task_slave_set_data

El uso de esta directiva dentro del constructo task se debe a una característica implícita del modelo de implementación utilizado en llc y al uso de MPI como lenguaje paralelo. Estos hechos tienen como efecto colateral que, en general, las variables "recuerdan" el valor que obtuvieron en la última ejecución. La ejecución de las tareas paralelas es especialmente sensible a este hecho, ya que su cómputo puede depender del valor inicial de algunas variables, debiendo asegurarse que este valor inicial se restaura cada vez que se ejecuta una nueva tarea.

Por este motivo, para los datos iniciales con valor fijo de las tareas, se ha añadido la directiva task_slave_set_data, que permite indicar a qué valores se deben inicializar las variables especificadas. Esta inicialización

se producirá cada vez que se lleve a cabo la ejecución de una nueva tarea. El uso de esta directiva no implica ninguna comunicación, sino que el compilador de `llc` se encarga de generar el código necesario e insertarlo al principio de cada tarea para que se lleve a cabo la inicialización. La sintaxis de esta directiva es:

```
#pragma llc task_slave_data(p, n, val {, p, n, val}*)
```

1. `p`: puntero al principio de la variable a inicializar.
2. `n`: número de elementos a inicializar a partir de la dirección indicada por `p` (se permite la inicialización tanto de variables simples como la inicialización parcial o total de vectores).
3. `val`: valor al que se inicializarán las variables especificadas.

2.8.2.6. Directiva `task_reduce_slave`

Esta directiva es usada en el constructo `task` para indicar que a los resultados enviados por el esclavo, el maestro debe aplicarle la operación de reducción especificada. La sintaxis de esta directiva se presenta a continuación:

```
#pragma llc task_reduce_slave (p_d, p_s, n, op {, p_d, p_s, n, op}*)
```

donde los parámetros son los siguientes:

1. `p_d`: puntero a la variable que contiene las contribuciones parciales de los esclavos.
2. `p_s`: puntero a la variable en la que el maestro almacenará la solución final. Esta variable debe ser diferente de `p_d`.
3. `n`: `llc` permite realizar operaciones de reducción sobre variables simples o bien total o parcialmente sobre vectores, llevando a cabo la operación elemento a elemento. Mediante el parámetro `n` el usuario indica el número de elementos sobre los que se efectuará la operación. empezando en la dirección indicada por los punteros.

4. op: Este parámetro especifica qué operación de reducción se aplicará. Esta operación puede ser una de las predeterminadas de llc mostradas en la tabla 2.1 (página 70) o bien una definida por el usuario.

Como puede observarse, la sintaxis y función de esta directiva es similar la directiva reduce explicada para el constructo for (entre otros) y que puede consultarse para más detalles en el apartado 2.5.2.2 (página 64).

Sin embargo, a diferencia de otras directivas de reducción estudiadas, el uso de esta directiva en el modelo de cola de tareas no implica que el maestro difunda el resultado global obtenido entre los esclavos. Para que se lleve a cabo esta difusión el usuario deberá especificar la correspondiente directiva taskq_master_result en el constructo taskq.

2.8.2.7. Directiva task_t_reduce_slave

La directiva task_t_reduce_slave es una variante de la directiva anteriormente estudiada task_reduce_slave. La diferencia entre ambas directivas es que con task_t_reduce_slave no es necesario indicar una segunda variable donde se almacenará la solución final. En su lugar se debe especificar el tipo de la variable a la que se le aplicará la operación de reducción. La sintaxis de esta directiva es similar a task_reduce_slave, salvando la diferencia indicada:

```
#pragma llc task_t_reduce_slave (p, t, n, op {, p, t, n, op}*)
```

donde los parámetros son los siguientes:

1. p: puntero a la variable que contiene las contribuciones parciales en los esclavos y que en el maestro contendrá la solución final tras aplicar la operación de reducción.

2. t: Tipo de datos de la variable sobre la que se realiza la operación de reducción.

3. n: llc permite realizar operaciones de reducción sobre variables simples o bien totalmente o parcialmente sobre vectores, llevando a cabo la operación elemento a elemento. Mediante el parámetro n el usuario indica el número de elementos sobre los que se efectuará la operación empezando en la dirección indicada por los punteros.

4. `op`: Este parámetro especifica qué operación de reducción se aplicará. Esta operación puede ser una de las predeterminadas de `llc` mostradas en la tabla 2.1 (página 70) o bien una definida por el usuario.

2.8.2.8. Directiva `taskq_barrier`

El uso de esta directiva en un punto del código fuerza a que la ejecución quede detenida en dicho punto hasta que todas las tareas que están siendo ejecutadas finalicen. Cuando esto sucede, la ejecución se reanuda.

La directiva `taskq_barrier` debe ser usada dentro de la región especificada por el constructo `taskq`, pero fuera de los bloques de código de tareas indicados por medio del constructo `task`. Su sintaxis es la siguiente:

```
#pragma llc taskq_barrier
```

2.8.3. Resumen de la sintaxis del constructo `taskq`

A continuación se recogen las diferentes directivas y su sintaxis:

`#pragma` [`intel`] `omp` [`parallel`$^{(a)}$] `taskq` {*omp_taskq_clause*$^{(b)}$}*
{`#pragma llc taskq_master_result` (*p, n* {*, p, n*}*)*
taskq_code$^{(c)}$

Las tareas deben especificarse dentro del código *taskq_code* por medio del constructo `task`. La declaración de cada tarea debe tener la siguiente sintaxis:

`#pragma` [`intel`] `omp task` {*omp_taskq_clause*$^{(b)}$}*
{`#pragma llc task_master_data` (*p, n* {*, p, n*}*) |
`#pragma llc task_slave_set_data` (*p, n, val* {*, p, n, val*}*) |
`#pragma llc task_slave_data` (*p, n* {*, p, n*}*) |
`#pragma llc task_slave_rnc_result` (*p, n, m, r* {*, p, n, m, r*}*) |
`#pragma llc task_reduce_slave` ({*p_data, ptr_sol, num_items, op*}*) |
`#pragma llc task_t_reduce_slave` (*p, t, n, op* {*, p, t, n, op*}*) }*
task_code

$^{(a)}$: Si la palabra clave `parallel` no se especifica, el constructo debe estar dentro de una región paralela.
$^{(b)}$: Todas las cláusulas `taskq` y `task` propuestas por `KAI-Intel` para `OpenMP` son admitidas por razones de compatibilidad. Sin embargo, no tendrán efecto en `llc` y para aquellos casos donde es necesario modificar los atributos de las variables compartidas, el usuario deberá usar las correspondientes directivas `llc` en vez de

las cláusulas de `KAI-Intel`.
(c): El código encerrado dentro de *taskq_code* y fuera de *task_code* puede incluir una o varias instancias la directiva `llc taskq_barrier` para forzar una sincronización entre todos los procesos.

2.9. Constructo `master`

Al programar en memoria compartida se debe tener en cuenta que diferentes threads pueden ejecutar el mismo código accediendo al mismo espacio de memoria. En estos casos, hay situaciones en los que se debe asegurar que un único thread ejecuta una determinada porción de código para evitar que se produzcan accesos concurrentes, interbloqueos, etc. Mediante esta directiva en `OpenMP` se asegura que un único thread (en este caso el thread maestro de cada equipo) ejecutará el código.

2.9.1. Ámbito de aplicación del constructo `master` en `llc`

En `llc`, el constructo `master` realmente no representa ningún modelo de paralelismo, sino que se ha incluido en `llc` por razones de compatibilidad con `OpenMP`. Como se indicó, el uso de este constructo toma sentido cuando se programa en memoria compartida y se debe asegurar que es un único thread el que actualiza una posición de memoria para evitar accesos concurrentes, etc.

Sin embargo, `llc` ha sido diseñado para ejecutar sobre un modelo de memoria distribuida, por lo que esta directiva no tiene sentido, ya que cada proceso dispone de una memoria privada y el acceso concurrente no es posible. De este modo, este constructo tal y como es usado en `OpenMP` no tiene efecto en el código producido por `llc`. Para que el programa generado por el compilador de `llc` tenga el mismo comportamiento que el que da lugar `OpenMP`, todos los procesos deberán ejecutar el bloque de código indicado. Este es el comportamiento por defecto de `llc` frente al uso de este constructo, que puede ser modificado usando las correspondientes directivas `llc`. La directiva `master` tiene la siguiente sintaxis:

```
#pragma omp master
```

2.9.2. Directivas `llc` del constructo `master`

A pesar de las consideraciones realizadas en el subapartado anterior, hay situaciones en las que al programar con `llc` interesa que sólo un proceso ejecute una porción de código, como, por ejemplo, en operaciones de entrada y salida, como imprimir un mensaje por pantalla, leer o escribir un fichero, etc. Para estos casos, `llc` provee una directiva que restringe a un solo proceso la ejecución del bloque de código marcado por el constructo.

No obstante, para el caso específico de operaciones de entrada y salida, `llc` dispone de un conjunto de macros que realizan estas funciones de una forma más simple (ver el apartado 2.16 en la página 102). Las directivas de `llc` que se pueden utilizar en este constructo se citan a continuación.

2.9.2.1. Directiva `onlymaster`

Esta directiva es propia de `llc` y su uso opcional dentro de un constructo `master` (o `single`) modifica el comportamiento de `llc`, asegurando que sólo el proceso maestro ejecutará el bloque de código delimitado dicho constructo. Su sintaxis es la siguiente:

```
#pragma llc onlymaster
```

2.9.2.2. Directiva `result`

En el constructo `master` (o `single`) se utiliza con el fin de difundir los resultados que el procesador maestro puede haber obtenido tras haber ejecutado en solitario el bloque de código señalado por el constructo.

Esta directiva tiene la sintaxis y significado detallado en el apartado 2.5.2.1 (página 63).

2.9.3. Resumen de la sintaxis del constructo `master`

A continuación se muestra la sintaxis del constructo `master` y las directivas permitidas en `llc`:

```
 #pragma omp master
[#pragma llc onlymaster]
{#pragma llc llc_result (p, n {, p, n}*)*
   code
```

2.10. Constructo single

El constructo single de OpenMP tiene el mismo comportamiento que el constructo master, salvo que en esta ocasión se asegura que un único thread ejecutará el código, pero no se especifica qué thread será, por lo que puede tratarse del thread maestro o de cualquier otro que pertenezca al equipo.

En cuanto a llc se refiere, este constructo es tratado de igual forma que el constructo master y todas las consideraciones que se presentaron en el apartado 2.9 de la página 93 son válidas, incluidas las posibles directivas y sintaxis, que se resume a continuación.

2.10.1. Resumen de la sintaxis del constructo single

La sintaxis del constructo single se presenta a continuación:

```
#pragma omp single {omp_single_clauses}*
[#pragma llc onlymaster]
{#pragma llc llc_result (p, n {, p, n}*)*
   code
```

2.11. Constructo barrier

El objetivo de este constructo es el de establecer un punto de sincronización explícito que todos los procesadores del grupo deberán alcanzar antes de que a cualquiera de ellos se le permita poder proseguir con la ejecución.

El constructo barrier no tiene cláusulas OpenMP ni directivas llc asociadas y su sintaxis es la siguiente:

```
#pragma omp barrier
```

2.12. Resumen de uso de directivas y constructos de llc

La tabla 2.3 ofrece un resumen del uso de las directivas de llc en los diferentes constructos de este lenguaje. Para entender esta tabla es necesario

tener en cuenta las siguientes notas:

- **(1)**: Los constructos `master` y `single` se representan juntos ya que en `llc` reciben el mismo tratamiento. El constructo `barrier` no se muestra puesto que no admite ninguna directiva.
- **(2)**: Se permite el uso alternativo de la cláusula de `OpenMP` `lastprivate`.
- **(3)**: Esta directiva de `llc` debe usarse tan sólo cuando complementa la cláusula `reduction` de `OpenMP`.

directiva\constructo	`for`	`sections`	`section`	`pipeline`	`taskq`	`task`	`m/s`[1]
`result`	√		√	√			√
`rnc_result`	√						
`nc_result`	√						
`lastresult`[2]	√	√					
`reduce`	√	√					
`reduction_type`[3]	√	√					
`weight`	√		√				
`send`				√			
`receive`				√			
`taskq_master_result`					√		
`task_master_data`						√	
`task_slave_data`						√	
`task_slave_set_data`						√	
`task_slave_rnc_result`						√	
`task_reduce_slave`						√	
`task_t_reduce_slave`						√	
`onlymaster`							√

Tabla 2.3: Directivas que aceptan los constructos de `llc`.

2.13. Paralelismo anidado y recursivo en `llc`

`llc` permite paralelismo multinivel eficiente mediante la anidación o recursividad en los constructos que definen bucles (constructo `for`), secciones (constructos `sections`/`section`) y segmentación paralela

(constructo pipeline). El paralelismo anidado o recursivo no se encuentra implementando en el modelo de cola de tareas (taskq/task), mientras que para el resto de constructos no tiene sentido su aplicación (master, slave y barrier).

El tratamiento del paralelismo multinivel se realiza de acuerdo con los fundamentos del modelo OTOSP: cada vez que en la ejecución de un constructo paralelo se encuentra un nuevo constructo, se realiza una división del grupo actual de procesadores en varios subgrupos, creándose un nuevo nivel de paralelismo que dará respuesta al constructo hallado.

En teoría, este proceso de creación de un nuevo nivel paralelo podría realizarse tantas veces como nuevos constructos paralelos anidados o recursivos se encuentren en el código. Sin embargo, existe un límite en la profundidad de niveles que se pueden alcanzar, que depende del número de procesadores que forman el grupo actual. En general, mientras haya más de un procesador en el grupo actual es posible realizar una nueva subdivisión y descender otro nivel de paralelismo, pero cuando el grupo esté formado por un solo procesador ya no será posible realizar nuevas divisiones, siendo el código afectado ejecutado secuencialmente por el único procesador disponible, ignorándose cualquier nuevo constructo paralelo que se encuentre en dicho código.

2.14. Resumen de compatibilidad entre llc y OpenMP

En su versión actual, llc es totalmente compatible a nivel léxico y sintáctico con el estándar 2.5 de OpenMP, reconociendo su gramática al completo. Para algunas directivas y cláusulas de OpenMP esta compatibilidad también se produce a nivel de significado y funcionamiento, mientras que otras no tienen significado o éste no es exactamente el mismo en llc. Esto se debe a que ambos lenguajes se basan en modelos diferentes, lo que implica que no todos los constructos, directivas y cláusulas tengan sentido en los dos modelos y, por lo tanto, no se pueda establecer una equivalencia total entre ambos lenguajes.

La compatibilidad que llc presenta a OpenMP se ha implementado con la finalidad de que el usuario tuviera que realizar el número mínimo de modificaciones para obtener un programa llc partiendo de un código OpenMP válido, y para que se conservara la validez del código OpenMP tras añadir las directivas de llc que fuesen necesarias.

2.14.1. Resumen de correspondencia entre constructos de OpenMP y llc

La tabla 2.4 presenta un resumen de los constructos del lenguaje OpenMP en su estándar 2.5, con una breve descripción e indicando si tienen o no correspondencia en llc. El lector interesado puede consultar una explicación y sintaxis detallada de los constructos de OpenMP en [123]. Para entender esta tabla deben tenerse en cuenta las siguientes notas:

- (1): El constructo/directiva de OpenMP no tiene sentido en el modelo de llc y/o bien su traducción consiste en omitirlo.
- (2): A pesar de que el constructo/directiva de OpenMP podría tener sentido en llc, éste no tiene una especial relevancia para los objetivos que se persiguen en este trabajo y, por lo tanto, no se ha considerado oportuno que fuera objeto de implementación en llc.
- (3): Los constructos indicados son los propuestos por el grupo KAI-Intel y no pertenecen al estándar de OpenMP 3.0.

2.14.2. Resumen de correspondencia entre cláusulas OpenMP y directivas llc

La tabla 2.5 recoge un resumen de las diferentes cláusulas del lenguaje OpenMP en su estándar 2.5, con una breve descripción e indicando si tienen o no correspondencia en llc. El lector interesado puede consultar una explicación y sintaxis detallada de las cláusulas de OpenMP en [123]. Las siguientes notas deben ser consideradas para el análisis de esta tabla:

- (1): La cláusula de OpenMP no tiene sentido en el modelo de llc y/o bien su traducción consiste en omitirlo.
- (2): A pesar de que la cláusula de OpenMP podría tener sentido en llc, éste no tiene una especial relevancia para los objetivos que se persiguen en este trabajo y, por lo tanto, no se ha considerado oportuno que fuera objeto de implementación en llc.
- (3): Para la cláusula reduction, el usuario deberá utilizar la directiva de llc reduction_type para complementar esta cláusula de OpenMP
- (4): La cláusula captureprivate no es conforme al estándar de OpenMP 3.0.

Constructo	Descripción	llc
parallel	Especifica un bloque de código que será ejecutado por múltiples threads.	Sí
for	Indica un bucle for cuyas iteraciones serán ejecutadas en paralelo.	Sí
parallel for	Especifica una región paralela que contiene una única directiva for.	Sí
sections	Delimita un bloque de código en el cual se localizan las secciones paralelas.	Sí
parallel sections	Indica una región paralela que contiene una única directiva sections.	Sí
section	Determina el bloque de código que constituye una sección paralela.	Sí
single	Indica un bloque de código que será ejecutada por un único thread.	Sí
master	Indica un bloque de código que sólo será ejecutado por el thread principal	Sí
critical	Especifica un bloque de código que sólo será ejecutado por un thread cada vez (no se permiten ejecuciones simultáneas).	No(1)
barrier	Sincroniza un equipo de threads.	Sí
atomic	Especifica que una dirección de memoria debe ser actualizada automáticamente, antes que permitir múltiples threads intentando escribir en ella.	No(1)
flush	Identifica un punto de sincronización en el que se debe asegurar una visión consistente de la memoria.	No(1)
ordered	Es usado para obtener una salida, por ejemplo, impresión en pantalla, que mantiene el orden secuencial en una ejecución paralela.	No(2)
threadprivate	La directiva threadprivate indica que deben replicarse aquellas entidades de ámbito global, de modo que cada thread tenga su propia copia.	No(2)
taskq(3)	Delimita un bloque de código en el cual se localizan las tareas paralelas.	Sí
parallel taskq(3)	Indica una región paralela que contiene una única directiva *taskq*	Sí
task(3)	Determina el bloque de código que constituye cada tarea paralela.	Sí

Tabla 2.4: Constructos de OpenMP y su correspondencia con llc

Cláusula	**Descripción**	`llc`
`private`	Declaran variables privadas a cada thread.	No(1)
`shared`	Declaran variables compartidas por todos los threads.	No(1)
`default`	Permite que el usuario especifique el carácter por defecto (compartido o ninguno) de las variables (sólo puede especificarse una de estas cláusulas por región paralela).	No(1)
`firstprivate`	Indica que las variables, además de ser declaradas como privadas, serán inicializadas automáticamente.	No(1)
`lastprivate`	Indica que las variables, además de ser declaradas como privadas, tendrán una copia del valor obtenido en la última ejecución de un bucle paralelo o en la última sección paralela.	Sí
`reduction`	Realiza una operación de reducción.	Sí(3)
`schedule`	Indica cómo serán dividas las iteraciones de un bucle (estáticamente, dinámicamente, guiada o en tiempo de ejecución) y, opcionalmente, el tamaño de las divisiones, si procede.	No(2)
`ordered`	Secuencializa y ordena el código dentro de la región que delimita.	No(2)
`nowait`	Indica que las *trheads* no deben sincronizarse al final de un bucle paralelo.	No(2)
`if`	Se evalúa una expresión y, si da un valor verdadero, un equipo de threads será creado. En otro caso, la región se ejecutará serialmente por el thread principal (sólo se permite especificar una cláusula).	No(2)
`copyin`	Provee de un medio para asignar el mismo valor a las variables declaradas con `threadprivate` en todas los threads del conjunto.	No(1)
`copyprivate`	Provee de un medio para asignar el mismo valor a las variables declaradas con `threadprivate` en todas los threads del conjunto.	No(2)
`num_threads`	Limita el número de threads.	No(1)
`captureprivate`(4)	Crea una copia privada.	No(1)

Tabla 2.5: Cláusulas de `OpenMP` y su descripción

2.14.3. Resumen del uso de cláusulas en los constructos de OpenMP

Para completar la información anterior, en la tabla 2.6 se expone información del uso de las cláusulas de OpenMP en los diferentes constructos de este lenguaje. Sobre esta tabla hay que realizar las siguientes aclaraciones previas:

- Los constructos que no admiten cláusulas no están recogidos en esta tabla.
- La marca $\surd$ indica las cláusulas que pueden ser usadas sin restricciones en los correspondientes directivas de OpenMP.
- La marca Δ identifica aquellas cláusulas que sólo se permiten si la correspondiente directiva de OpenMP, además de indicar el constructo, abre una región paralela (mediante el uso de `parallel`, p. ej: `#pragma omp parallel for`).
- La marca ∇ especifica que esta cláusula no puede ser usada si la correspondiente directiva deOpenMP, además de indicar el constructo, abre una región paralela (mediante el uso de `parallel`, p. ej: `#pragma omp for`).
- Los constructos `taskq` y `task` y la directiva `captureprivate` no son conformes al estándar de OpenMP 3.0.

2.15. Funciones OpenMP en llc

Por compatibilidad con OpenMP, llc admite el uso de algunas de las funciones de OpenMP más comunes. En concreto se permite el uso de las rutinas relacionadas con la obtención del número e identificador de threads de OpenMP (en llc corresponde con procesadores). Para un listado completo de las funciones de OpenMP, el lector puede consultar su estándar 2.5 en [123].

Al detectarse alguna de estas funciones de OpenMP soportadas, el compilador de llc generará el código adecuado para que el programa llc tenga un comportamiento equivalente al de OpenMP. Sin embargo, se recomienda no usar este tipo de funciones si no es estrictamente necesario, ya que podría perderse la compatibilidad con el programa secuencial.

cláus/direct	`parallel`	`for`	`sections`	`single`	`taskq`	`task`
`private`	√	√	√	√	√	√
`shared`	√	Δ	Δ		Δ	
`default`	√	Δ	Δ		Δ	
`firstprivate`	√	√	√	√	√	
`lastprivate`		√	√		√	
`reduction`	√	√	√		√	
`schedule`		√				
`ordered`		√			√	
`nowait`		∇	∇	√	∇	
`if`	√	Δ	Δ		Δ	
`copyin`	√	Δ	Δ		Δ	
`copyprivate`				√		
`num_threads`	√	Δ	Δ		Δ	
`captureprivate`						√

Tabla 2.6: Cláusulas que aceptan los constructos de `OpenMP`.

A continuación se presenta la lista de funciones `OpenMP` soportadas. Aquellas funciones en cursiva no pertenecen a `OpenMP`, sino que han sido incluidas en `llc`. Puede consultarse una información más detallada del significado de estas funciones en la explicación de su traducción, concretamente en la tabla 3.3 (página 145).

- `omp_get_thread_num()`
- `omp_get_num_threads()`
- `omp_get_num_procs()`
- *`omp_get_global_thread_num()`*
- *`omp_get_global_num_threads()`*

2.16. Macros `llc`

`llc` incorpora una serie de macros que pueden servir de ayuda al usuario a la hora de realizar operaciones de entrada/salida u operaciones relacionadas con ficheros, como la creación, apertura o cierre de los mismos. El uso de estas macros en los códigos `llc` tiene las siguientes características:

- **Operaciones de Entrada (E)**: Sólo el procesador maestro realizará la operación de entrada de datos, difundiendo luego estos datos a todos los restantes procesadores del grupo.

- **Operaciones de Salida (S) o de manejo de ficheros (F)**: Sólo el procesador maestro realizará dicha operación

Macro `llc`	**Op**	**Eq.** `ANSI C`	**Sintaxis** `llc`
`LLC_fread()`	E	`fread()`	*Idem que* ***ANSI C***
`LLC_fwrite()`	S	`fwrite()`	*Idem que* ***ANSI C***
`LLC_read()`	E	`read()`	*Idem que* ***ANSI C***
`LLC_write()`	S	`write()`	*Idem que* ***ANSI C***
`LLC_fscanf()`	E	`fscanf()`	`LLC_fscanf(fd,fmt,ptr)`
`LLC_fscanf4()`	E	`fscanf()`	`LLC_fscanf4(fd,fmt,arg1,arg2)`
`LLC_fscanf5()`	E	`fscanf()`	`LLC_fscanf5(fd,fmt,arg1,arg2,arg3)`
`LLC_fscanf6()`	E	`fscanf()`	`LLC_fscanf6(fd,fmt,arg1,arg2,arg3,arg4)`
`LLC_fprintf()`	S	`fprintf()`	*Idem que* ***ANSI C***
`LLC_scanf()`	E	`scanf()`	`LLC_scanf(fd,fmt,ptr)`
`LLC_printf()`	S	`printf()`	*Idem que* ***ANSI C***
`LLC_printMaster()`	S	`printf()`	*Idem que* ***ANSI C***
`LLC_fopen()`	F	`fopen()`	*Idem que* ***ANSI C***
`LLC_fclose()`	F	`fclose()`	*Idem que* ***ANSI C***
`LLC_open()`	F	`open()`	`LLC_open(path,flags)`
`LLC_creat()`	F	`creat()`	*Idem que* ***ANSI C***
`LLC_close()`	F	`close()`	*Idem que* ***ANSI C***

Tabla 2.7: Macros de `llc` para tratamiento de operaciones de *Entrada/Salida* y gestión de ficheros

La tabla 2.7 muestra las macros de `llc`, indicando para cada una de ellas el tipo de operación de los mencionados (entrada, salida o tratamiento de ficheros), la función equivalente en `ANSI C` y la sintaxis de `llc`, especificando si se usa la misma de `ANSI C` o tiene sintaxis propia.

Estas macros se suministran con el fin de facilitar la programación al usuario, pero éste debe entender que la utilización de las mismas podría limitar la compatibilidad del código de `llc` con la versión secuencial y la de `OpenMP` correspondiente. Si se está interesado en conservar esta compatibilidad, no se recomienda su uso salvo que sea indispensable.

2.17. Introducción de código MPI en llc y otras consideraciones

Además del uso de las funciones de OpenMP o las variables llc mencionadas en la tabla 3.3 y de las macros de llc recogidas en la tabla 2.7, el programador puede interaccionar con llc escribiendo directamente código MPI.

Sin embargo, el usuario debe tomar las precauciones necesarias que aseguren que el código MPI introducido no afectará ni entrará en conflicto con el código producido por el compilador de llc. En particular, el usuario deberá tener en cuenta lo siguiente:

- El código generado por llc inserta automáticamente llamadas a MPI_Init() y MPI_Finalize(). El usuario no debería incluir llamadas a ninguna de estas funciones.
- Debe tenerse en cuenta que cada procesador tiene en todo momento un identificador dentro del grupo al que pertenece actualmente (generalmente indicado por el comunicador llc_CurrentGroup) y otro identificador global referido al comunicador MPI_COMM_WORLD. Esto debe tenerse presente en especial a la hora de realizar comunicaciones entre los distintos procesadores. En la tabla 3.3 (página 145) se muestran las variables de llc que permiten conocer para cada procesador el identificador y cantidad de procesadores en el grupo local y actual.
- El uso de llamadas a funciones MPI produce, en general, una pérdida de compatibilidad con la versión secuencial y OpenMP. Esto debe ser tenido en cuenta en caso de estar interesado en conservar esta triple compatibilidad (secuencial, OpenMP y llc) en el mismo código.

Por último, indicar que llc sigue el convenio de utilizar como prefijo las cadenas llc_ y LLC_ en los identificadores de variables, funciones, macros, etc., que son introducidos durante la fase de generación de código. El usuario debe evitar utilizar cualquier identificador con estos prefijos a la hora de desarrollar su aplicación para no entrar en conflicto con el código generado por el compilador de llc.

Capítulo 3

`llCoMP`: un compilador para `llc`

3.1. Introducción

Una vez presentado `llc`, este capítulo se centra en el estudio de `llCoMP`, el compilador que da soporte a nuestro lenguaje. `llCoMP` es un *compilador-traductor* [1] que toma como entrada un código fuente `ANSI C` anotado con directivas de `llc` (y `OpenMP`) y produce como salida un código fuente paralelo escrito en `ANSI C` con llamadas a funciones de `MPI`. A continuación se citan las principales características de `llCoMP`:

- *Eficiencia del código generado*: `llCoMP` ha sido diseñado con el objetivo de generar un código paralelo eficiente, que presente un rendimiento comparable al de otros lenguajes y compiladores paralelos similares.
- *Portabilidad del compilador*: Nuestro compilador no está diseñado para ninguna arquitectura específica y es portable a cualquier máquina donde se encuentren las siguientes herramientas:
  - Generador de analizadores léxicos (`flex`): Necesario para producir el analizador léxico
  - Generador de analizadores sintácticos (`YACC`/`Bison`): Necesario para generar el analizador sintáctico
  - Compilador de `C`: Necesario para la generación de `llCoMP`
- *Portabilidad del código generado*: Aún más importante que la portabilidad del compilador es la del código generado. Al producir el compilador código fuente y no un fichero ejecutable, el ámbito de

aplicación de la salida de compilador serán todas las arquitecturas y sistemas donde esté disponible un compilador de ANSI C y la librería MPI.

En llCoMP la portabilidad del código producido siempre ha sido un punto crítico y el compilador se ha diseñado para que el código de salida no tenga ninguna dependencia con la máquina en la que se generó. Esto deriva en otra ventaja, un código de llc se puede compilar en una determinada máquina y luego transferir a cualquier otro equipo para obtener el ejecutable enlazando con las librerías de MPI, sin que sea necesario que llCoMP esté instalado en este último equipo. La figura 3.1

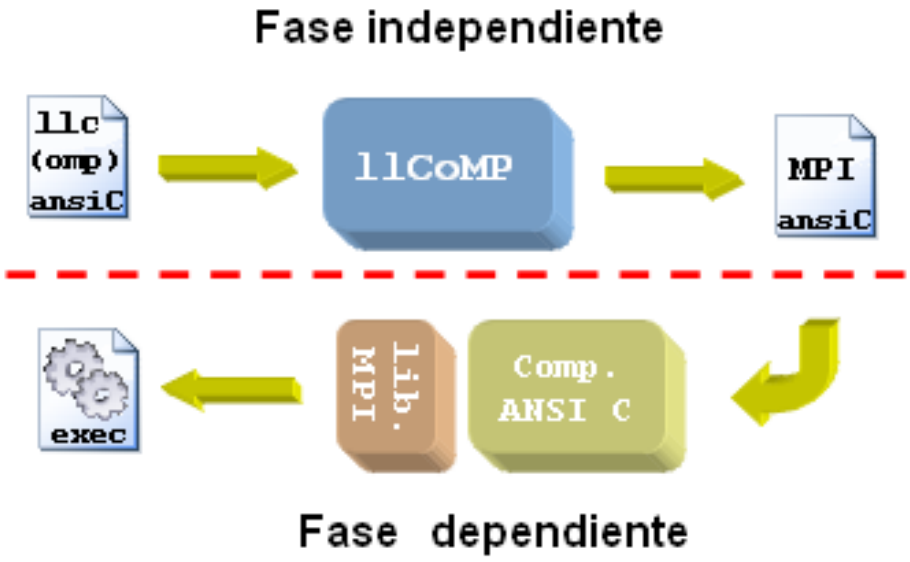

Figura 3.1: Proceso de generación de un ejecutable a partir de un código fuente llc

muestra un esquema general de las distintas etapas de compilación que, por defecto, se le aplican a un código fuente llc. Existen dos fases diferenciadas: una primera independiente del sistema, en la que la entrada y salida son códigos fuentes, y una segunda que producirá el binario ejecutable y que, por lo tanto, es dependiente del sistema en el que se generó.

- *Facilidad de desarrollo y disponibilidad*: En la implementación del compilador hemos tratado de separar en lo posible el proceso de compilación de los recursos utilizados para generar el código paralelo final. El objetivo que se busca es crear una independencia que permita

modificar el código paralelo generado sin interferir en los procesos de análisis realizados por el compilador y viceversa.

Es posible obtener el código fuente del compilador a través del sitio web del lenguaje [100].

3.2. Diseño del compilador

El diseño e implementación de `llCoMP` ha seguido un proceso continuo de evolución hasta alcanzar su estado actual. En sus orígenes [102, 67], `llCoMP` fue concebido como una serie de *macros* cuyo uso emulaba con bastantes restricciones a algunos de los constructos y directivas del compilador actual. Esta idea se desechó al quedar patente las limitaciones del modelo a medida que los requisitos iban aumentando en número y complejidad. Por otro lado, el uso de las macros no permitía compatibilizar de forma simple la versión secuencial y la paralela en un mismo código.

Del desarrollo inicial de `llCoMP` mediante macros se pasó al estudio de otras posibilidades que dieran soporte a nuestro lenguaje. La idea de desarrollar un compilador completo que produjera código ejecutable fue desestimada porque el esfuerzo que hubiera sido necesario invertir para la implementación de la fase de generación de código no estaría compensado por ninguna mejora sustancial en el resultado obtenido, además de limitar la portabilidad de nuestro desarrollo. De este modo, finalmente se eligió como mejor solución el enfoque de *compilador-traductor* que ha sido implementado.

Para el desarrollo de `llCoMP` se han tomado como referencia otros productos o trabajos de investigación que resolvían una problemática similar. Entre ellos cabe realizar una mención especial a `Cilk` [33]. El compilador de `Cilk` fue originalmente desarrollado por R. Miller para su tesis doctoral [110] en el M.I.T.

`Cilk` es definido por sus desarrolladores como un lenguaje de programación paralela usando *multithreading* basado en `ANSI C`, mientras que `cilk2c` [21] es el preprocesador con comprobación de tipos usado para la traducción del lenguaje `Cilk` a `C`. `cilk2c` acepta el lenguaje `Cilk` (un superconjunto de `ANSI C`) y genera un código `C` portable. Este enfoque es similar al que sigue `llCoMP` y `cilk2c` ha servido de punto de referencia para el desarrollo de nuestro compilador.

3.3. Fases de la compilación

La fase de análisis [97] de llCoMP consta de un *análisis léxico* y un *análisis sintáctico*. Además de éstos, llCoMP cuenta con una etapa previa de *preprocesado*. A continuación se explican con detalle cada una de estas tareas.

3.3.1. Preprocesado

Durante el análisis léxico y sintáctico es necesario disponer de una lista de los identificadores de todos los tipos de datos usados en el código de entrada. Para obtener esta lista no basta con analizar el fichero de entrada, ya que éste puede contener directivas de inclusión (`#include`) que incluyan ficheros externos en donde se declaren los tipos de datos o que contengan más directivas `#include`.

Para resolver este problema y otros relacionados, muchos compiladores realizan un paso previo de *preprocesado* [1]. Durante esta etapa se procesan los archivos a incluir, así como otras operaciones tales como expansión de macros y constantes, etc. El resultado de este proceso es un fichero único "autocontenido" en el sentido que no contiene ninguna referencia a código de ficheros externos.

llCoMP realiza esta tarea de preprocesado invocando por defecto al preprocesador de C. El resultado de este proceso se recoge en un fichero temporal sobre el cual actuarán las etapas posteriores. Se construye así una *tabla de símbolos* simplificada que contiene los identificadores de todos los tipos de datos utilizados, así como información extra como el fichero y número de línea donde se encuentra cada declaración. Estos datos serán útiles para la gestión de errores y avisos durante el proceso de compilación, permitiendo indicar la posición exacta donde se detectó el error. Para una consulta más rápida, llCoMP implementa esta tabla de símbolos como una *tabla hash*.

La necesidad de realizar un preprocesado previo añade mayor complejidad al diseño del compilador, debido a los efectos colaterales que esta etapa introduce. Los principales problemas que se derivan de este proceso son el *código específico* y el *código que no cumple con el estándar*. A continuación definiremos estos problemas, así como la solución adoptada.

3.3.1.1. Inclusión de código específico

El resultado de la expansión de macros e inclusión de ficheros suele contener código específico de la máquina en la que ha sido preprocesado y, por tanto, no portable a otras máquinas. Este hecho rompe la filosofía de portabilidad que deben seguir los códigos generados por `llCoMP`.

La salida obtenida del preprocesador es analizada tanto léxica como sintácticamente para obtener el listado completo de los tipos, pero si se desea garantizar la portabilidad, la etapa de generación de código no debe basarse en la salida preprocesada. En `llCoMP` hemos resuelto este problema aplicando una política de doble compilación o *compilación en dos pasadas*. Las acciones desarrolladas en cada una de estas pasadas se resumen a continuación.

1. *Pasada 1*: Se preprocesa el fichero fuente y se construye su tabla de símbolos.

2. *Pasada 2*: Una vez construida la tabla de símbolos en la pasada anterior, la segunda pasada del compilador se realiza directamente sobre el código original. De este modo, tanto el análisis léxico como el sintáctico se aplicarán sobre el fichero fuente primitivo, generando el fichero de salida y asegurando su portabilidad[1].

El realizar dos pasadas implica que el tiempo de compilación se vea incrementado. Si el usuario no está interesado en la portabilidad y desea realizar una única pasada, el compilador dispone de un argumento en línea de comando para ello, `-noport`[2]. Al usar esta opción, el compilador realizará una única pasada y la compilación y generación de código tomará como base el código preprocesado. El resultado producido será, en la mayoría de casos, no portable y sólo se podrá usar en la máquina en la que se llevó a cabo la compilación.

Si lo desea, el usuario también podrá realizar únicamente la segunda pasada del compilador, deshabilitando el preprocesado. Para ello puede hacer uso del argumento `-nopre` al invocar a `llCoMP`. En esta situación, el compilador sólo buscará los tipos dentro del código fuente especificado

[1] La portabilidad del código generado por `llCoMP` es la misma que la correspondiente al código de entrada. Si el usuario usó en su código original alguna declaración, macro, función, etc. *dependiente* de una máquina en cuestión, el código de salida, obviamente, conservará esa dependencia.

[2] El lector puede consultar la tabla 3.1 de la página 118 para acceder al listado completo de opciones disponibles en `llCoMP`

por el usuario. Esta opción podría aplicarse cuando el código del usuario sólo usa tipos predefinidos en ANSI C o bien tipos definidos por el usuario a partir de éstos. Si el compilador detecta cualquier otro tipo, se producirá e informará del error.

En la práctica se ha demostrado que el tiempo de compilación que llCoMP consume es en la mayoría de los casos de apenas unos pocos segundos, por lo que la diferencia entre que el compilador realice una o dos pasadas es prácticamente imperceptible para el usuario. Por este motivo el comportamiento por defecto es el de la compilación en dos pasadas.

3.3.1.2. Inclusión de código no estándar

El preprocesado también introduce una serie de problemas que llCoMP ha tenido que resolver para generar la salida final. Entre el resto de inconvenientes que hemos encontrado durante esta etapa, destaca la inclusión que algunos compiladores de C realizan de código que no cumple el estándar ANSI C a pesar de ser invocado con los argumentos que fuerzan la compatibilidad.

La mayor complicación en este sentido que podemos destacar son las *extensiones* del compilador de gcc. Las extensiones son un código no estándar (y por lo tanto, no portable) que algunos compiladores incluyen para aumentar el nivel de optimización y ofrecer funcionalidades adicionales [65].

Como ejemplo, se puede citar la función typeof() de gcc. El uso de esta función nos permite obtener el tipo de datos con el que se ha declarado una variable, lo cual podría ser de utilidad para la implementación de un compilador, pero su uso no es posible con llCoMP ya que, de usarlo, sólo podría compilarse en una máquina donde una versión de gcc estuviera disponible y fuese compatible con esta extensión.

El problema con las extensiones es que no están recogidas en el estándar de ANSI C y, aunque el usuario no haga uso de ellas, suelen estar localizadas en alguno de los ficheros del sistema que se incluyen durante el preprocesado. Por este motivo, al realizarse el análisis léxico o sintáctico durante la primera etapa de compilación, llCoMP detecta este código no estándar como errores.

Una posible solución a este problema sería desechar el uso de compiladores que no sean estándar. Sin embargo, el hecho de que gcc estuviera dentro de este grupo desaconsejaba esta opción, debido a que este compilador es uno de los más extendidos y usados, prácticamente disponible en cualquier distribución y arquitectura.

La solución que se tomó fue la de preparar `llCoMP` para que fuese flexible y permitiera el análisis de las extensiones no estándar más comunes. El análisis de este código no estándar se diseñó relajando las restricciones y tolerando una mayor flexibilidad, lo que nos permite estar preparados en un futuro a las inclusiones o modificaciones de extensiones que se produzcan en nuevas versiones de los compiladores afectados. De este modo se reduce el efecto negativo que este hecho produciría, si bien esto crea una cierta dependencia que no es posible eliminar. Hemos comprobado que otras aplicaciones similares a la nuestra, como pueden ser la nombrada `Cilk` o la utilidad `cxref` [41] también adolecen de esta característica.

Además del problema con las extensiones, existen otras situaciones en las que no se sigue el estándar `ANSI C`, lo que produce conflictos. Como resumen de estos problemas que han aparecido durante la etapa de preprocesado y que han tenido que ser resueltos en `llCoMP` con estrategias similares a las descritas, podríamos citar:

- Uso de tipos y macros `built-in`, que no se encuentran definidas en el código, sino internamente en el compilador.
- Uso de identificadores de tipos para nombrar un campo de una estructura o unión.
- Uso de identificadores de tipos como etiquetas para saltos con `goto`.
- Uso del operador `'?'` con un solo argumento: `expr '?' ':' expr`
- Expresiones de conversiones de tipo o *typecast* con sintaxis: `'(' TYPEname ')' '{' expr'}'`
- Existencia de bloques estructurados de código entre paréntesis.
- Aparición de nuevas palabras reservadas con sus correspondientes tokens y sintaxis asociada:
  - `extension`
  - `inline`
  - `restrict`
  - `attribute`
  - `alignof`
  - `typeof`
  - `asm`

Como queda patente, el preprocesado es una fase crítica, por lo que llCoMP ofrece diversas opciones que, al ser invocadas en línea de comandos, pueden modificar el comportamiento del preprocesador, adaptándolo a las necesidades particulares del usuario. Un listado completo de las opciones puede consultarse en la tabla 3.1 de la página 118.

3.3.2. Análisis léxico

El analizador léxico de llCoMP se ha desarrollado haciendo uso de flex[3] [71], herramienta para la generación de analizadores léxicos o programas que reconocen patrones léxicos en un texto.

La principal función del analizador léxico es la de reconocer los símbolos terminales de la gramática o *tokens*. Además, gestionará sus atributos, que generalmente consisten en el lexema asociado. Para la descripción del analizador léxico, flex utiliza un emparejamiento de expresiones regulares y código C, denominadas *reglas*, acompañadas de rutinas C de apoyo. Cada vez que se detecta una cadena que concuerda con alguna de estas expresiones regulares, se ejecuta el código C asociado.

El analizador léxico implementado soporta al completo el reconocimiento de ANSI C. A partir de este lenguaje base, se ha añadido el reconocimiento de OpenMP y el de llc. Algunas estadísticas del analizador léxico de llCoMP son las siguientes:

- más de 140 tokens
- más de 200 reglas
- más de 2400 estados NFA y 1250 DFA (más de 9800 palabras)
- más de 66000 transiciones

Las tareas a desarrollar por el analizador léxico difieren según en la pasada en la que se encuentre:

1. *Primera pasada*: Se crea la tabla de símbolos que contiene los identificadores de los tipos de datos no predefinidos, a partir del preprocesado del fichero fuente.

[3] En muchos sistemas, el analizador léxico de llCoMP también es compatible con lex

2. *Segunda pasada*: Se realiza un análisis léxico completo tomando como entrada el fichero fuente indicado por el usuario. Cada vez que el analizador léxico encuentra un identificador, realizará una búsqueda del lexema en la tabla de símbolos y dependiendo del resultado de esta búsqueda devolverá el token *nombre de tipo definido*(`TYPEDEFname`) en caso de búsqueda exitosa o el token *identificador* (`IDENTIFIER`) si el elemento a buscar no es encontrado en la tabla.

La parte más sensible del analizador léxico es aquella que reconoce las directivas de compilación con las que el usuario expresa el paralelismo (`#pragma ...`), ya sean de `llc` u `OpenMP`. El funcionamiento del compilador varía en gran medida dependiendo de si el análisis de la entrada se está realizando dentro o fuera del ámbito de una de estas directivas.

- El código exterior a estas regiones corresponde con la parte secuencial del programa y la generación de código fuente que el compilador realiza se basa en gran medida en una copia del código de entrada.
- En el interior del ámbito de las directivas paralelas el compilador toma una especial relevancia al generar el código paralelo que se insertará en la salida.

Para diferenciar entre estas dos situaciones, el analizador léxico utiliza un estado *inclusivo* y otro *exclusivo* [71] según el tipo de directiva encontrada. Cuando es detectada una de las directivas de `OpenMP` o `llc`, se entra en uno de los estados exclusivos y se devuelven los tokens específicos de las regiones paralelas, que pueden ser diferentes a los de la partes secuenciales del código[4].

El estado exclusivo de directiva paralela comienza al ser detectada una directiva de `llc` o de `OpenMP`. Para una mayor facilidad al programador y para minimizar las modificaciones a introducir si se parte de un código `OpenMP`, se permite el uso indistinto de los identificadores `llc` y `omp` en las directivas. El estado exclusivo se abandonará cuando la directiva finalice, lo cual se produce cuando se encuentra un retorno de carro, siempre que éste no esté precedido por el carácter \, que significaría que la directiva continúa en la línea siguiente.

Otra de las tareas del analizador léxico de `llCoMP` es la del tratamiento de los comentarios y de separadores como espacios en blanco, tabuladores,

[4] Un ejemplo de esto lo tenemos en la palabra reservada `for`: generalmente, al encontrar este lexema en el código, el analizador léxico devolverá el token de palabra reservada de bucle, pero si lo encontramos dentro de una directiva del tipo `#pragma omp for ...`, se devolverá el token que indica el inicio de una región paralela de tipo *forall*.

retornos de carro, etc. Mediante el uso de otro estado exclusivo, el analizador léxico descarta los comentarios, incluso si estos están anidados. Se ha pretendido que el código de salida del compilador sea lo más legible posible y sea conforme con unos ciertos criterios de formato estético, entendiendo éste como el uso coherente de indentaciones, espacios, líneas en blanco, etc. Por este motivo, durante la etapa de generación de código se tienen en cuenta algunos criterios de embellecimiento de código.

Para facilitar el proceso de desarrollo del compilador, el analizador léxico ha sido preparado para gestionar y producir mensajes de depuración en tiempo de ejecución. Mediante algunas de sus rutinas es posible realizar una traza del proceso de análisis y mostrar qué token se está analizando y el valor de su atributo, si procede. Se ha tenido en cuenta el hecho de que alguno de estos atributos no son imprimibles, mostrando en ese caso su representación hexadecimal y el código del carácter especial al que representa.

El analizador léxico también lleva a cabo la gestión de la línea y fichero que se está analizando en cada caso, ya sea el resultado del preprocesado durante la primera pasada o bien del código fuente del usuario durante la segunda.

3.3.3. Análisis sintáctico

El analizador sintáctico de `llCoMP` ha sido desarrollado haciendo uso de la herramienta `Bison` [70], un generador de analizadores sintácticos de propósito general. Esta herramienta convierte una descripción gramatical para una gramática independiente del contexto *LALR(1)* en una aplicación `C` que analiza dicha gramática. La gramática que hemos diseñado es compatible con `YACC` [90], por lo que se puede usar indistintamente cualquiera de ambas herramientas.

El analizador sintáctico implementado en `llCoMP` reconoce la gramática de `ANSI C` [97]. Por compatibilidad, el analizador también implementa la gramática de `OpenMP` para `C` acorde con su estándar 2.5 [123]. A partir de estas dos gramáticas se han añadido las reglas necesarias para dar soporte a `llc`. A continuación se muestran algunas estadísticas para el analizador sintáctico finalmente implementado:

- más de 5500 líneas de código
- más de 750 reglas
- más de 1350 estados

- Sin conflictos de desplazamiento/reducción, salvo el inherente a la gramática de `ANSI C` derivado del uso de `else`, se resuelve ejecutando siempre un desplazamiento

Como el analizador léxico, la función que desarrolla el analizador sintáctico varía dependiendo de si se encuentra en la primera o en la segunda pasada del código. La labor que desempeña en cada pasada se resume a continuación:

1. *Primera pasada*: Junto con el analizador léxico, el analizador sintáctico construye la tabla de símbolos que contiene la información de los tipos no predefinidos. Durante esta fase la generación de código está desactivada y las únicas acciones que se llevan a cabo se producen cuando se detecta una definición de un tipo. Esto sólo ocurre en puntos muy concretos del código de entrada donde es posible la definición de nuevos tipos, usando para ello la palabra reservada `TYPEDEF`.

2. *Segunda pasada*: Se comprueba que el código de entrada es correcto sintácticamente y se aplican las reglas para generar el código de salida, sirviéndose de la información de la tabla de símbolos.

Entre otros aspectos a destacar del analizador sintáctico podemos mencionar la *política de recuperación de errores* que ha sido implementada mediante la inclusión de "*reglas de producción de error*". Haciendo el uso del token `error` y la macro `yyerrok` se han introducido producciones que contemplan los errores más comunes, de modo que el compilador pueda recuperarse en caso de producirse algún error. Se pretende así prolongar el análisis, evitando en lo posible que un sólo error genere una cascada de errores que aborte el proceso.

Además de errores, `llCoMP` también contempla una política de generación de *avisos* (*warnings*) clasificados en varios niveles según su importancia. Esto permite al usuario poder indicar al compilador qué tipos de avisos quiere recibir al invocar el compilador, por medio de un argumento que se especifica por línea de comandos.

Al notificar un error o aviso, el compilador, además del código y descripción del error, muestra en qué fichero, línea y el atributo del token más cercano, con el fin de facilitar al usuario su identificación y solución. Para mejorar este aspecto de identificación de errores en el fichero de salida, durante el análisis se lleva un registro del punto exacto del fichero en el cuál se está y, durante la generación de código, el compilador añade directivas `#line` que referencian al fichero original de entrada.

3.3.4. Analizador semántico

La inclusión de un analizador semántico en llCoMP que llevara a cabo, entre otras funciones, una comprobación de tipos, fue objeto de debate durante el proceso de diseño y desarrollo del compilador.

Sin embargo, varios fueron los motivos por los cuales esta etapa no fue añadida al compilador:

- Se deseaba que el código del compilador fuese lo más sencillo y manejable posible, ya que el camino natural del compilador es el de seguir creciendo en cuanto al número de constructos paralelos implementados que den soporte a nuevas formas de explotar paralelismo. Añadir un análisis semántico supone aumentar considerablemente la complejidad del proceso de compilado, dificultando el mantenimiento y modificación del mismo.
- El esfuerzo de añadir un analizador semántico no queda compensando por el beneficio que éste generaría. Hay que mencionar que la salida generada por llCoMP deberá ser tratada por un compilador de ANSI C que lleve a cabo el proceso de enlazado con las rutinas de las librerías de MPI. Así pues, cualquier error semántico que llCoMP no haya identificado, le será notificado al usuario en este momento. En general, este proceso de compilación doble se realiza de forma encadenada, actuando primero llCoMP y, si no se detectan errores, luego el compilador de ANSI C. De este modo, el usuario recibirá los errores conjuntamente y referidos siempre al fichero que indicó como entrada, careciendo de importancia qué compilador fue el que detectó e informó sobre dichos errores (véase la figura 3.1 en página 106).
- Por último, como ya se ha comentado, el objetivo de este trabajo no es el de construir un compilador completo, sino una herramienta que dé soporte al lenguaje de llc, demuestre su validez y permita la traducción de códigos. En este sentido, la implementación de un analizador sintáctico no aporta una ventaja sustancial.

3.3.5. Opciones del compilador

llCoMP soporta diferentes argumentos que, al ser usados en línea de comandos, modifican su comportamiento para adaptarse a diferentes situaciones y requerimientos de los usuarios. Seguidamente se recogen las diferentes opciones:

NOMBRE

`llCoMP`

SINOPSIS

`llCoMP` [*opciones*] [*fichero*]

DESCRIPCIÓN

`llCoMP` analiza el *fichero* especificado como un programa `ANSI C` que puede contener directivas de `llc` y/o `OpenMP`. A partir de esta entrada `llCoMP` genera un programa paralelo en `C` que incluye llamadas a la librería de `MPI` en el *fichero.llc.c*. Si se omite *fichero*, usa la entrada y salida estándar.

OPCIONES

Los argumentos en línea de comandos de `llCoMP` se recogen en la tabla 3.1

3.4. Generación de código

`llCoMP` no genera un código intermedio como tal, sino que durante el proceso de análisis va llevando a cabo la traducción del código de entrada al código paralelo de salida. La traducción se realizará de modo diferente si el código se encuentra dentro del ámbito de una directiva paralela o en el exterior:

1. *Generación de código pasiva*: Durante la generación pasiva se realiza una copia literal del código de entrada en la salida. Este proceso se desarrolla generalmente en aquellas regiones del código de entrada que no se encuentran dentro del ámbito de las directivas de compilador de `llc`, si bien existen algunas excepciones.

2. *Generación de código activa*: En esta forma de generación de código, `llCoMP` transforma activamente la entrada para generar una salida paralela. La generación activa de código se lleva a cabo dentro del ámbito de los constructos paralelos, pero también en algunos puntos que requieren transformación en el código secuencial de entrada, como la función `main`, funciones de `llc` y algunas de `OpenMP`, macros de `llc`, etc.

Este apartado se centra en el estudio del proceso de generación activa de código que realiza `llCoMP`, basada en la utilización de *patrones de código*.

Opción	Descripción
	Opciones generales
-help	Imprime ayuda de uso
-options	Imprime todas las opciones
-copy	Imprime la información de copyright
-v	Imprime la información de la versión
	Opciones de salida
-o <fich>	Escribe la salida en *fich*
-N	No emite directivas de líneas
	Opciones de aviso
-ansi	Deshabilita las *extensiones* de GCC y elimina la definición de __GNUC__
-W<n>	Asigna el nivel de avisos; n está entre 1-5. (por defecto n=4)
-Wall	Igual que -W5
-il	Ignora las directivas de línea (utiliza el número de línea actual)
-name <x>	Usa *stdin* con x como fichero en los mensajes
	Opciones de preprocesado
-nopre	No preprocesa
-P<str>	Asigna el comando del preprocesador a *str*
-pre	Imprime el comando del preprocesador y los modificadores
-I<path>	Especifica la ruta (*path*) para buscar los ficheros a incluir
-Dmacro[=value]	Define una *macro* (con un valor opcional)
-Umacro	Elimina la definición de una *macro*
-H	Imprime el nombre de cada fichero de cabecera
-undef	No predefine *macros* no estándar
-nostdinc	No analiza los ficheros estándar a incluir
-noport	Emite una salida específica expandiendo las directivas de preprocesado (salida no portable)
	Opciones de depuración
-lex	Muestra *tokens* léxicos
-yydebug	Realiza una *traza* de las acciones y estado de la pila del analizador sintáctico
	Opciones de compatibilidad con CC
--	Ignora las opciones desconocidas (para compatibilidad del Makefile con cc)

Tabla 3.1: Opciones del compilador llc

3.4.1. Patrones de código

Si se realiza un estudio de un amplio conjunto de aplicaciones paralelas, es posible concluir que la mayoría de las mismas pueden agruparse en un número reducido de conjuntos atendiendo al paradigma usado para lograr el paralelismo [34].

Un estudio más profundo de estas aplicaciones paralelas desvela que existe un alto porcentaje de código común en las mismas, tal como distribución de los procesadores, comunicaciones, etc. Este código depende en gran medida del paradigma utilizado y no de la aplicación en sí. Acompañando a este código común existe un código específico de cada aplicación, que recoge aquellas operaciones concretas y específicas, como tratamiento de datos, etc.

La idea que subyace a la implementación de `llCoMP` es que, una vez identificado el paradigma que definirá el paralelismo, existe un código que da soporte general a dicho paradigma y que es independiente de la aplicación que se desea paralelizar. Al ser este código fijo para cada paradigma, nos referiremos a él como *código estático*.

El código estático conforma un esqueleto sobre el que se construirá la aplicación paralela final, añadiendo para ello el código específico dependiente de la aplicación, que denominaremos *código dinámico*.

En este sentido, `llCoMP` basa su traducción en la utilización de *patrones de código* que permiten que el compilador sea capaz de generar el código estático y el código dinámico y entrelazarlo de forma correcta para obtener la aplicación paralela final.

3.4.1.1. Patrones estáticos

Este tipo de patrones contienen el código paralelo fijo que se tomará como base para la generación de la salida. La elección del patrón estático depende del tipo de paralelismo que el usuario haya indicado mediante el uso de directivas. Las funciones típicas de este código son las relacionadas con la gestión del paralelismo, como la inicialización de los recursos paralelos, la distribución y equilibrado de las tareas, gestión del entorno para la ejecución de las mismas, comunicación de los datos y resultados y liberación de los recursos utilizados al finalizar.

Dependiendo de la complejidad del constructo, éste puede ser dividido en varias etapas y cada etapa estar almacenada en un patrón diferente. Para algunas etapas existe más de un patrón estático, ya que hemos seguido

la política de implementar el patrón del caso general, pero también crear patrones optimizados para los casos particulares más comunes. Esto da lugar a una familia de patrones entre la cual el compilador selecciona la mejor combinación de forma transparente al usuario, basándose en la información de las directivas paralelas que éste especificó.

Los patrones estáticos se almacenan en ficheros de texto plano que contienen el código paralelo tal y como será incluido en la aplicación final. Este código incluye unas etiquetas que indican la posición en la cual el compilador deberá incluir los argumentos especificados por el usuario al usar el constructo y/o directivas. Las etiquetas tienen la sintaxis `$n`, donde `n` es el índice del argumento, empezando por 0.

Cada patrón estático lleva asociado una constante que indica el número de argumentos que debe recibir. Estos argumentos se copian en un vector durante el análisis sintáctico y en la generación de código el compilador sustituirá cada elemento `$i` por el valor de la posición i-ésima del vector de argumentos.

Además de las etiquetas de los argumentos, los patrones estáticos contienen etiquetas que marcan la posición en la que el compilador debe incluir el código de los *patrones dinámicos*. El formato de estas etiquetas es `$#m#nombre_fich`, donde `nombre_fich` es el nombre del fichero temporal que almacena el patrón dinámico y `m` es la profundidad de indentación que el compilador tendrá en cuenta al insertar el código del patrón dinámico, respetando los criterios de embellecimiento de código.

En el listado 3.1 se presenta un ejemplo de patrón estático, concretamente un fragmento del código de comunicación de resultados en un pipeline. El patrón recibe tres argumentos (`$0`, `$1` y `$2`) que son usados en las líneas 12 y 19. En la línea 13 encontramos otra etiqueta que indica al compilador que en esa posición debe introducir el código del patrón dinámico contenido en el fichero temporal `.forall_comm_res_init.llc.tmp`, utilizando para ello 4 niveles de indentación. De la misma forma, en la línea 20 debe introducir el código del fichero temporal `.forall_comm_res_pack.llc.tmp`, esta vez con 5 niveles de indentación.

Los patrones estáticos se almacenan en un repositorio de ficheros de texto plano independientes del compilador. Esta separación de los códigos estáticos del propio compilador posibilita un diseño más simple y ágil, y contribuye a mejorar el mantenimiento y optimización de los patrones. De este modo, los cambios que se lleven a cabo en el compilador no alteran los patrones, mientras que los patrones pueden ser desarrollados directamente como si se estuviera codificando la aplicación paralela, sin interferencia con

```
{
/ *
  * Argumentos de entrada $i:
  * $0 => Variable del bucle
  * $1 => Valor inicial de la variable del bucle
  * $2 => Valor final de la variable del bucle
  */

  ...

  llc_size_tmp = 0;
  for ($0=($1)+LLC_NAME; $0<=($2); $0+=LLC_NUMPROCESSORS) {
    $#4#.forall_comm_res_init.llc.tmp
  }

  if (llc_size_tmp > 0) {
    LLC_CHECK_MALLOC(llc_buf, llc_size_tmp, char);
    llc_buf_ptr = llc_buf;
    for ($0=($1)+LLC_NAME;$0<=($2);$0+=LLC_NUMPROCESSORS) {
      $#5#.forall_comm_res_pack.llc.tmp
    }
  }
  ...
}
```

Listado 3.1: Fragmento de un patrón estático de los pipelines

el compilador.

3.4.1.2. Patrones dinámicos

Los patrones dinámicos contienen el código que gestiona aquellos aspectos específicos de la aplicación, relacionados principalmente con operaciones secuenciales de tratamiento de datos, entre los que destaca la gestión de los buffers de comunicación de datos y las operaciones de optimización, empaquetado y desempaquetado de datos, etc.

Al tratarse de código específico, estos patrones son generados durante la compilación de la aplicación y el código se almacena en ficheros temporales. Luego este código será incluido en las posiciones especificadas por las correspondientes etiquetas del patrón estático, para generar así el código de salida final.

3.5. Implementación de constructos

En este apartado se revisan aspectos relativos a la implementación que `llCoMP` realiza de los distintos constructos y su traducción al código paralelo final.

3.5.1. Constructos `master` y `single`

Como ya se mencionó en el apartado 2.9 (página 93), el código anotado con estas dos directivas debe ser ejecutado en todos los procesadores para poder asegurar la consistencia de memoria según el modelo OTOSP. Sin embargo, el usuario puede forzar que sólo el proceso master ejecute el código asociado (p. ej., para operaciones de E/S), usando para ello la directiva de `llc onlymaster`. La traducción que realiza `llCoMP` de estos dos constructos difiere según se use o no esta directiva.

1. *No se utiliza la directiva* ***`onlymaster`***: La traducción es pasiva, copiando directamente en la salida el código que en el origen estaba englobado en el constructo.

2. *Se utiliza la directiva* ***`onlymaster`***: En este caso se controla que sólo el procesador maestro del grupo ejecute este código. Si el usuario ha utilizado directivas `result`, los resultados indicados se transmitirán a

los demás procesadores del grupo mediante llamadas a la función de `MPI_Bcast()` de `MPI`.

3.5.2. Constructo `for`

El constructo `for` da soporte a los bucles paralelos, uno de los métodos más usados para la paralelización de aplicaciones científicas dadas sus características. Es por esto que los lenguajes paralelos suelen dar un amplio soporte a estos bucles. Para evitar confusiones con la palabra reservada de `C` para los bucles, también nos referiremos a este constructo como `forall`.

En `llc` se permite la paralelización de bucles `for` de una forma similar a la realizada en `OpenMP`, aunque se permiten otras posibilidades como el uso de *bucles generales* (apartado 2.5.1.1 en página 61).

Hemos puesto un especial empeño en garantizar que el código de salida generado por el compilador abarque con un mayor nivel de optimización aquellas circunstancias particulares más comunes.

Para una gestión más fácil, la generación de código en los bucles `forall` se ha dividido en cuatro etapa bien diferenciadas. Para cada una de estas etapas existen diversos patrones de código que optimizan diferentes situaciones. `llCoMP` genera la salida eligiendo entre los patrones que mejor se adapten a las características del código y la información suministrada en las directivas, de forma transparente al usuario.

Puesto que otros constructos siguen la misma filosofía de implementación que en los bucles paralelos, compartiendo incluso algunas de sus etapas, en este apartado haremos un estudio más profundo de estas etapas para luego tomarlas como referencia al analizar otros constructos.

3.5.2.1. Inicialización

Durante esta primera etapa se crean e inicializan las estructuras de datos que contendrán la información de control que se utilizará en las restantes etapas. Uno de los puntos más destacables de este etapa es que en ella se lleva a cabo la asignación y distribución de los procesadores y tareas a ejecutar.

Cada iteración del bucle paralelo corresponde con una *tarea*. De este modo, los parámetros principales que rigen esta distribución son el número de procesadores disponibles en el grupo actual (nP) y el número de tareas a realizar (nT).

Existe un tercer parámetro que influye en la distribución: la información de pesos (W). Si este dato es indicado por el usuario, el algoritmo de repartición realizará una distribución proporcional a cada peso con respecto al total de la suma de todos los pesos, por lo que se requiere que los pesos sean números enteros positivos. Si no se indican los pesos, la asignación tratará de ser equitativa.

A efectos de implementación de la traducción, existe un cuarto parámetro a considerar: si se va a aplicar o no la optimización para bucles con incremento unitario positivo. Este último factor decidirá qué fichero de código estático será usado para la traducción, si el general o el optimizado.

Lo primera operación que debe realizarse en la etapa de inicialización es calcular los tres parámetros mencionados (nP, nT y W):

- El número de procesadores disponibles en el grupo actual (nP) no precisa ninguna operación en especial, ya que la variable `LLC_NUMPROCESSORS` almacena este valor.

- El número total de tareas (nT) o de iteraciones paralelas se calcula de forma distinta según se trate del caso general o del caso de bucle paralelo con incremento unitario positivo. En este último caso, nT se obtiene a partir de las expresiones y condiciones indicadas en el bucle ($nT = end - begin + 1$), mientras que para el caso general el número de iteraciones se calcula al ejecutar el bucle vacío.

- En caso de que el usuario especifique pesos para las tareas, se genera un vector de pesos W, en el que cada posición W_i coincide con el peso asociado a la tarea T_i, comprobando que todos estos valores sean enteros positivos. De igual modo, se calculará el sumatorio de todos los pesos ($\sum w_i$) y el peso medio ($\bar{w}$), valores que serán posteriormente usados en la asignación.

Una vez determinados nP, nT y W, se procederá a realizar la asignación de procesadores y tareas. Las distintas políticas de asignación se aplican en función de los valores de nP con respecto a nT:

$nP = 1$: Si el grupo actual sólo dispone de un procesador, no se aplica ninguna política de asignación, ya que el único procesador disponible ejecutará todas las tareas secuencialmente. Se está en el caso `LLC_SEQUENTIAL`.

$nP < nT$: Si hay más tareas a ejecutar que procesadores, se aplica una *política de asignación de bloques de tareas* a cada procesador, por lo que la situación se denomina `LLC_BLOCK`.

$nP > nT$: Si, por el contrario existen más procesadores que tareas a ejecutar, los procesadores se agruparán en tantos grupos como tareas haya, y cada uno de estos grupos ejecutará la misma tarea. De este modo se aplica una *política de asignación de grupos de procesadores* y el caso se denomina `LLC_GROUPS`.

$nP = nT$: Si el número de procesadores y tareas coincide, cada procesador ejecutará una tarea. Esta situación es un caso particular de los dos anteriores y se puede escoger cualquiera de las mismas. Por defecto, `llCoMP` trata esta situación como si fuera un caso `LLC_BLOCK`.

Una vez identificado el caso adecuado para la situación actual , éste se almacena para luego ser utilizado en las etapas de ejecución, comunicación y finalización con el fin de determinar qué criterios aplicar.

A la hora de presentar en esta memoria las políticas y su funcionamiento interno, no se han utilizado los nombres empleados en el código, sino que haremos uso de otros más cortos y representativos dada la situación, con el fin de facilitar la explicación. La tabla 3.2 establece una relación entre los nombres simbólicos empleados al definir las políticas y el nombre utilizado en el código, indicando en cuál de las dos políticas son usadas (**B**: bloques y **G**: grupos) y añadiendo una breve descripción. Esta tabla será de gran ayuda si se desea consultar los patrones usados para la generación de código.

Simb.	Nombre código.	B	G	Descripción
nT	`llc_nT`	√	√	Número de tareas
nP	`LLC_NUMPROCESSORS`	√	√	Número de procesadores
V	`llc_F`	√	√	Vector de información
W	`llc_W`	√	√	Vector de pesos
$\bar{w}$	`llc_mW`	√		Peso medio de las tareas
$\sum W_i$	`llc_nW`		√	Peso total de las tareas
wp	`llc_w`	√		Peso asignado a un procesador
T_r	`llc_rnT`	√		Tareas restantes sin asignar
τ	`LLC_WEIGHT_TOL`	√		Factor de tolerancia
g	`llc_grp`		√	Grupo del procesador

Tabla 3.2: Nombres simbólicos empleados al definir las políticas, su equivalencia con los usados en la implementación y una breve descripción

Política de asignación de tareas (`LLC_BLOCK`): Esta política se aplica cuando el número de tareas es mayor o igual que el de procesadores ($nT \geq$

nP) y su objetivo es el de asignar de forma controlada y eficiente las distintas tareas entre los procesadores, mediante bloques de tareas consecutivas.

Para ello, se crea un vector V que guardará la información del bloque de tareas asignados a cada procesador, de modo que el procesador i ($0 \leq i < nP$) ejecutará el rango de tareas determinado por $[V_i, V_{i+1} - 1]$.

Si la información de los pesos está disponible, se utilizará el vector de pesos W y el peso medio ($\bar{w}$), anteriormente calculados, para distribuir las tareas en bloques. La distribución se realiza de modo que el peso total de cada uno de ellos se aproxime lo más posible al peso medio ($\bar{w}$), respetando la restricción de que cada procesador debe ejecutar al menos una tarea. Si, por contrario, el usuario no indicó pesos, la distribución se realizará de forma equitativa, asignando similar número de tareas a cada procesador. Aplicar el algoritmo exacto que resuelva este problema podría ser muy costoso, por lo que `llCoMP` aplica una heurística.

Política de asignación de procesadores (`LLC_GROUPS`): La estrategia de asignación de procesadores implementada siempre asigna una tarea a cada uno de los procesadores, lo que implica que, al haber más procesadores que tareas, en algunos casos más de un procesador ejecutará la misma tarea.

Todos estos procesadores que ejecutan la misma tarea pertenecen al mismo *grupo*, lo que aporta las siguientes ventajas:

- Al final de la ejecución paralela todos los procesadores deben poseer los datos modificados por el resto de procesadores. Al ejecutar la misma tarea todos los procesadores de un mismo grupo, no será necesario realizar comunicaciones intergrupales, sino que todas las comunicaciones se producen entre procesadores de diferentes grupos.

- Puesto que todos los procesadores de un mismo grupo han ejecutado la misma tarea, todos dispondrán de los datos que se debe comunicar, por lo que la comunicación podrá realizarse en paralelo entre los procesadores de diferentes grupos.

- Se permite *paralelismo multinivel*. En los grupos con más de un procesador, todos los procesadores que lo formen estarán trabajando sobre los mismos datos, por lo que, a su vez, pueden volver a dividirse en subgrupos más pequeños tantas veces como sea posible.

Al igual que la política anterior, la información de los grupos de procesadores se guarda en un vector V, pero en esta vez referido a las tareas,

indicando que la tarea i-ésima será ejecutada por los procesadores del rango $[V_i, V_{i+1} - 1]$.

Si el usuario ha indicado pesos, se utilizan los datos previamente calculados de los pesos (almacenados en el vector W) y el peso total ($\sum W_i$). El objetivo es intentar que a toda las tareas se le asignen tantos procesadores consecutivos como sea necesario para igualar la proporción $\frac{nT_i}{nT}$ con $\frac{W_i}{W}$, asegurando que a cada procesador se le asigna al menos una tarea. Como en el caso anterior, aplicar el algoritmo exacto para este caso puede ser muy costoso, por lo que se aplica una heurística.

3.5.2.2. Ejecución

Dependiendo de la política de asignación empleada en la etapa de inicialización el proceso de ejecución diferirá. A continuación se presentan las tareas más relevantes a realizar en esta etapa dependiendo de la situación actual.

- $nP = 1$ (`LLC_SEQUENTIAL`):

  El único procesador disponible ejecutará todas las tareas, es decir, el código de todas las iteraciones.

- $nP \leq nT$ (`LLC_BLOCK`):

  En este caso cada procesador ejecutará una o más tareas según indique el vector V calculado en la etapa de inicialización.

  Antes de proceder a la ejecución hay que asignar a la variable del bucle paralelo el valor correcto de la iteración actual, a la vez que las variables que indican el número de procesadores (`LLC_NUMPROCESSORS`) y el identificador del procesador dentro del grupo actual (`LLC_NAME`) son modificadas para simular durante la ejecución que el procesador pertenece a un grupo virtual unitario (`LLC_NUMPROCESSORS = 1` y `LLC_NAME = 0`).

  Antes de realizar esta modificación se salva el entorno de ejecución actual, que luego será restaurado al finalizar esta etapa.

- $nP > nT$ (`LLC_GROUPS`):

  Si existen más procesadores que tareas, el grupo actual se divide para dar lugar a nuevos subgrupos de procesadores que, a su vez, son susceptibles de volver a repetir el proceso de división (paralelismo anidado), por lo que el entorno a gestionar es más complejo.

El proceso de ejecución es el siguiente:

1. Se calcula el número de procesadores total del nuevo grupo
2. Se calcula el identificador del procesador en el nuevo grupo
3. Se salva el entorno de ejecución actual
4. Se realiza una llamada a la función de MPI `MPI_Comm_split()` para crear el nuevo grupo
5. Se accede a la tarea (iteración) que le corresponde al procesador y se ejecuta el código asignado
6. Se restaura el entorno de ejecución previo
7. Se recuperan los valores originales del identificador (`LLC_NAME`) y número de procesadores del grupo (`LLC_NUMPROCESSORS`)
8. Se libera el grupo creado mediante una llamada a la función de MPI `MPI_Comm_free()`

3.5.2.3. Comunicación

La etapa de comunicaciones es la más compleja de los bucles paralelos, no sólo por el proceso en sí mismo, sino también por la diversidad de posibles escenarios que pueden darse. Para simplificar el estudio de esta etapa, se obviarán algunos detalles y se presentará una visión de conjunto de las situaciones más representativas.

- $nP = 1$ (`LLC_SEQUENTIAL`):

  Al haberse realizado la ejecución secuencialmente y existir un solo procesador, no se precisan comunicaciones.

- $nP \leq nT$ (`LLC_BLOCK`): Cuando existen más tareas que procesadores, varias tareas habrán sido ejecutadas por el mismo procesador. Aprovechando esta característica de la asignación por bloques, `llCoMP` establece una serie de operaciones para minimizar el número de comunicaciones, basándose en el empaquetamiento de datos. Para ello, se establecen varios niveles en los que se aplican estas operaciones.

  El primer nivel de optimización afecta a los datos a comunicar por cada tarea. Si es necesario realizar una comunicación múltiple (más de una variable o variables compuestas, como vectores), estos datos se empaquetarán creando un único mensaje, al que se le añadirá la información necesaria para que el receptor pueda llevar a cabo el

proceso de desempaquetado y copia en las variables de destino. Estos procesos de empaquetado y desempaquetado se llevan a cabo de modo transparente al usuario.

Gracias a este primer nivel de optimización, el número total de comunicaciones a llevar a cabo viene dado por la expresión $nT(nP-1)$, es decir, los datos de cada tarea se deben transmitir al resto de procesadores.

El siguiente nivel de optimización, realizado también de forma transparente al usuario, se produce en aquellos procesadores que han ejecutado más de una tarea. El mensaje que debe transmitir cada una de estas tareas será empaquetado en un nivel superior, generando al final del proceso un único mensaje por procesador que contiene todos los datos e información de desempaquetado.

De este modo, el número de comunicaciones a realizar se reduce a $nP(nP-1)$.

Las comunicaciones se realizan mediante la llamada a la función `MPI_Bcast()` de `MPI`. Cada procesador realizará una única llamada a esta función para difundir sus resultados al resto de procesadores, por lo que el proceso final de comunicación de datos implica un total de nP llamadas a la función `MPI_Bcast()`.

- $nP > nT$ (`LLC_GROUPS`):

  En el caso de comunicación con asignación por grupos, donde hay más procesadores que tareas, se dará la situación en la que varios procesadores habrán ejecutado una misma tarea. `llCoMP` aprovecha esta característica para realizar las comunicaciones de un modo más eficiente.

  En primer lugar se minimiza el número de comunicaciones por procesador, llevando a cabo un empaquetamiento de datos a nivel de tarea como el realizado en la asignación por bloques, con lo que se consigue que cada procesador sólo tenga que realizar una comunicación.

  El diagrama de comunicaciones para el caso de la distribución por grupos es más complejo que el de bloques. En este hecho influye la asimetría que puede existir en los grupos si han sido creados acorde con la información de pesos de las tareas.

  Para simplificar el proceso de comunicación, `llCoMP` establece una política de *procesadores colaboradores* entre los diferentes grupos. Un procesador de un grupo será *colaborador* de un procesador de otro

grupo si comparten el mismo identificador dentro de su respectivo grupo. Cuando no sea posible establecer una relación directa entre los procesadores, debido a la diferencia en el número de procesadores de los grupos, se aplicará una política de comunicación cíclica.

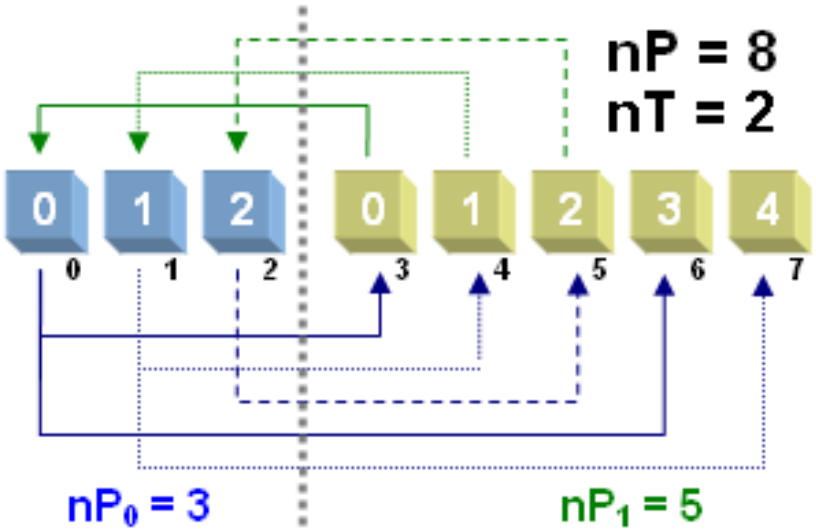

Figura 3.2: Comunicaciones realizadas en asignación de procesadores con dos grupos ($nP = 8$, $nT = 2$, $nP_0 = 3$ y $nP_1 = 5$)

La figura 3.2 muestra un ejemplo de esta situación. Un grupo original de $nP = 8$ procesadores se ha divido para ejecutar $nT = 2$ tareas, dando como resultado dos subgrupos. Debido a la diferencia de pesos de tareas, el primer grupo dispone de $nP_0 = 3$ procesadores y el segundo de $nP_1 = 5$ procesadores. El identificador que recibe cada procesador en el nuevo subgrupo se indica dentro de la caja que simboliza cada procesador, mientras que debajo de ella, en negro, se indica su identificador en el grupo original.

Una vez terminada la etapa de ejecución, las comunicaciones se realizan entre los *procesadores colaboradores* de cada grupo, de modo que el procesador 0 del primer grupo envía y recibe del procesador 0 del segundo grupo, el 1 con el 1 y así sucesivamente. Como el segundo grupo es mayor que el primero, las comunicaciones son cíclicas hasta transmitirse los datos a todos los procesadores, por lo que el procesador 0 del primer grupo también enviará al 3 del segundo, y el 1 al 4. Todas las comunicaciones son realizadas utilizando llamadas a las funciones `MPI_Send()` y `MPI_Recv()` de `MPI`.

En esta figura 3.2 quedan patentes las ventajas de esta asignación que ya fueron comentadas durante la etapa de inicialización: no existen comunicaciones dentro del mismo grupo, las comunicaciones entre los procesadores pueden realizarse en paralelo y cada uno de los grupos generados puede subdividirse a su vez para generar paralelismo multinivel.

- *Reducciones*

  El proceso de comunicación para las reducciones es una situación especial distinta de las ya presentadas. Por este motivo tiene su propio patrón de comunicaciones y será estudiado de forma independiente en este apartado.

  Las comunicaciones en el caso de las reducciones siguen un esquema fijo y se dividen en dos etapas.

  1. La primera de ellas consiste en el envío de los resultados parciales obtenidos por todos los procesadores al procesador maestro global (procesador con identificador 0 en el grupo global). En el caso de asignación por bloques, todos los procesadores enviarán los resultados parciales al maestro, mientras que en la asignación por grupos, sólo el procesador maestro de cada subgrupo (identificador 0 en el grupo local) hará el envío al maestro global.

     Todos estos envíos se realizan mediante una llamada a la función `MPI_Send()`, y serán recibidas por el maestro con el correspondiente `MPI_Recv()`.
  2. La segunda etapa tiene lugar cuando el procesador maestro ha realizado la operación de reducción y dispone de la solución total. En este punto el maestro difunde este resultado entre todos los demás procesadores para asegurar la coherencia de memoria. Esta difusión se realiza mediante una llamada a la función `MPI_Bcast()` y afecta a todos los procesadores, independientemente de la asignación por bloques o grupos.

  Como en otras situaciones, en caso de tener que comunicarse múltiples datos, éstos son empaquetados para reducir el número de comunicaciones.

3.5.2.4. Finalización

En la etapa de finalización se realiza la liberación de la memoria obtenida para las estructuras de gestión de las etapas previas, así como se restaura el

entorno previo antes de la entrada al bucle paralelo.

3.5.3. Constructo `sections`

El proceso de traducción de las secciones paralelas se realiza como un caso particular de los bucles paralelos. Para ello, se lleva a cabo un proceso de conversión en el que cada sección pasará a considerarse como una iteración de un bucle paralelo virtual. Una vez generado el código necesario para realizar esta transformación, el modo de llevar a cabo la traducción de las secciones paralelas es similar al de los bucles paralelos ya estudiados, si bien es necesario establecer algunas consideraciones adicionales.

Si se estudia el proceso de ejecución de las iteraciones de un bucle paralelo, la diferencia entre ejecutar una iteración u otra sólo reside en el valor de la variable del bucle, ya que las distintas iteraciones no pueden presentar dependencias entre sí. De este modo, si se quisiera ejecutar la iteración i-ésima sólo se debe asignar a la variable del bucle su valor i-ésimo y luego ejecutar el código especificado en el bucle.

Este mismo comportamiento puede obtenerse en el caso de las secciones, uniendo las distintas secciones para construir un gran `switch` y utilizando una variable (`llc_section`) que simula el comportamiento de la variable del bucle paralelo.

`llc` implementa esta conversión asignando a cada sección paralela un *identificador* (`<id_sect_act>`) que refleja el orden de aparición de cada sección paralela en el código. A partir de este identificador, el código a ejecutar de cada sección paralela se engloba entre sentencias `case <id_sect_act>:` y `break;`. El listado 3.2 recoge un ejemplo simple de secciones paralelas, mostrándose en el listado 3.3 su traducción a través del procedimiento indicado.

La traducción queda completada añadiendo el código de gestión de los bucles paralelos con su cuatro etapas, que ya han sido estudiadas en el apartado 3.5.2 (página 123), modificado pertinentemente para que tome `llc_section` como variable de bucle y tenga como espacio de iteraciones $[0, N-1]$, donde N es el número total de secciones paralelas. Como este bucle siempre tendrá un incremento unitario positivo, la traducción se realiza aplicando los patrones optimizados para esta situación.

Sin embargo, existen diferencias entre los bucles y secciones paralelas. La más importante de ellas reside en el ámbito de las directivas: mientras que en los bucles paralelos las directivas se indican en el ámbito del bucle y afectan

```
1  /* Código original */
2  #pragma omp sections
3  {
4  #pragma omp section
5    {
6      f0(A0);
7      g0(B0);
8    }
9  #pragma omp section
10   {
11     f1(A1);
12     g1(B1);
13   }
14 #pragma omp section
15   {
16     f2(A2);
17     g2(B2);
18   }
19 }
```

Listado 3.2: Ejemplo de secciones paralelas.

```
1  /* Traducción de las secciones */

3  switch (llc_section) {

5    case 0:
6        f0(A0);
7        g0(B0);
8        break;

10   case 1:
11       f1(A1);
12       g1(B1);
13       break;

15   case 2:
16       f2(A2);
17       g2(B2);
18       break;
19 }
```

Listado 3.3: Ejemplo de traducción para secciones paralelas.

a todas las iteraciones, en el caso de las secciones paralelas existen varias directivas que se especifican a nivel de cada sección y que sólo afectan a las variables de la misma.

Esta situación es común entre las directivas de comunicación de resultados. Un ejemplo lo tenemos con la directiva `result` (subapartado 2.6.2.1, página 77) que especifica qué variables se modifican en cada sección, por lo que debe ser usada localmente en cada una de las secciones para luego enviarse los resultados a los restantes procesadores.

`llCoMP` identifica las variables que deben ser comunicadas en cada una de las secciones a partir de la información de las directivas y añade el código necesario para compatibilizar estas situaciones con los patrones de comunicación de los bucles paralelos.

Esta consideración anterior también es extrapolable a los pesos indicados con la directiva `weight` (subapartado 2.6.2.3, página 79). Mientras que en los bucles paralelos se especificaban mediante una única expresión que afecta a todas las iteraciones, en las secciones se indican de forma opcional en cada sección. Si esta directiva es usada, a cada sección se le asignará el peso indicado si posee esta directiva o bien un peso por defecto de valor 1 en aquellas secciones que carezcan de ella. A partir de este punto, la distribución de procesadores o tareas, dependiendo del caso, se realiza como la estudiada en los bucles paralelos.

3.5.4. Constructo `pipeline`

El constructo pipeline permite ejecutar paralelismo de segmentación. La directiva `llc` para pipelines se basa en la construcción de los mismos a través de un bucle `for`, donde cada una de las iteraciones corresponde con una etapa paralela.

Esta articulación de los pipelines en torno a un bucle paralelo permite establecer una similitud entre los patrones ya estudiados y los del pipeline. En el constructo pipeline las etapas de *inicialización*, *ejecución* y *finalización* ya estudiadas en otros constructos previos están implementadas en un mismo patrón estático principal. Para las comunicaciones se hace uso de patrones auxiliares según el tipo de las mismas. A continuación se hará un estudio de cada uno de estos patrones.

3.5.4.1. Patrón principal

Este patrón realiza la gestión de las etapas de inicialización, ejecución y finalización del pipeline. Estas funciones son similares a sus correspondientes en los bucles paralelos, pero más simples debido a que los pipelines no poseen la diversidad de posibles escenarios de los bucles.

La primera operación que realiza este patrón es salvar el entorno de ejecución actual, lo que incluye el comunicador original y las variables que indican el número de procesadores del grupo y el identificador.

Seguidamente se realiza la inicialización del pipeline paralelo. La distribución de procesadores se lleva a cabo atendiendo al número de tareas (nT) y de procesadores (nP), de un modo similar al ya estudiado en otros constructos. Si existen más procesadores que tareas ($nP > nT$) se divide el comunicador actual mediante la función `MPI_Comm_split()` para crear subgrupos de procesadores con un reparto equitativo de tareas en lo posible. Para la situación alternativa ($nP \leq nT$), cada procesador ejecutará un número equitativo de etapas.

Una vez distribuidos los procesadores, se reparte el espacio de etapas del pipeline paralelo entre los mismos para ejecutar el código de cada una de ellas según la información obtenida en el paso anterior. Este espacio de etapas se distribuye asignando iteraciones del bucle que simula el pipeline a los distintos grupos de procesadores.

Una vez finalizada la ejecución, se procede a liberar las estructuras utilizadas y se restaura el estado previo salvado. Adicionalmente, si $nP > nT$ se liberan los comunicadores que habían sido creados para los subgrupos.

3.5.4.2. Patrones de comunicaciones

Las comunicaciones en un pipeline ya están establecidas y cada etapa sólo puede enviar datos a la etapa inmediatamente posterior y recibir de la inmediatamente anterior. Esto simplifica el proceso de comunicación de los pipelines. `llCoMP` traduce las comunicaciones mediante una serie de patrones estáticos auxiliares dependiendo del caso.

- **Envío de datos**: El patrón estático traduce la directiva `send` de `llc` (subapartado 2.7.3.1, página 83) por la llamada a la función `MPI_Send()`. Los datos de envío se obtienen de la directiva, mientras que el identificador del procesador actual es usado para determinar a qué etapa corresponde y cuál es la siguiente a la que se debe realizar

el envío. También se comprueba que la etapa actual no sea la última, caso en el que se descarta el envío.

- **Recepción de datos**: Se procesa la situación inversa a la anterior, traduciendo la directiva `receive` de `llc` (subapartado 2.7.3.2, página 83) por la correspondiente llamada a la función `MPI_Recv()`. Al igual que en el caso anterior, los datos se obtienen de la directiva, mientras que el identificador del procesador actual es usado para determinar a qué etapa corresponde y cuál es la anterior de la que se deben recibir los datos. Si nos encontramos en la primera etapa, no es posible recibir datos de etapas anteriores, por lo que la directiva se descarta.

- **Comunicación de resultados**: Este patrón lleva a cabo la traducción de la directiva `result` de `llc` (subapartado 2.7.3.3, página 84) , que difunde los resultados obtenidos por el pipeline a todos los procesadores.

  El resultado final de un pipeline generalmente se almacena en un vector o matriz sobre el cual actúa cada etapa. Tras ejecutar el pipeline paralelo, cada procesador o grupo de procesadores habrá ejecutado una o varias tareas, por lo cual cada uno de ellos tendrá el valor resultado de una porción de este vector que deberá transmitir al resto, a la vez que recibir aquellos datos de las tareas que no ha ejecutado.

  `llCoMP` prepara el proceso de comunicación de los resultados finales para que esta difusión se lleve a cabo mediante una llamada a la función `MPI_Allgatherv()`, lo que simplifica las comunicaciones con respecto al equivalente realizado con las funciones `MPI_Send()` y `MPI_Recv()`.

  Como en otras situaciones similares, las comunicaciones múltiples se empaquetan en un único mensaje para reducir el número total de las mismas. Para este proceso es necesario la generación de buffers de envío y recepción, generados por `llCoMP` de forma transparente al usuario mediante el uso de patrones dinámicos. Como el tamaño de estos buffers es variable, se hace necesario añadir una comunicación adicional mediante la función `MPI_Allgather()` para informar a los distintos procesadores del tamaño de datos que van a recibir.

3.5.5. Constructo `taskq`

`llCoMP` implementa el modelo de *colas de tareas (Workqueuing)* mediante una aproximación *maestro/esclavo* [130]. En esta implementación, un procesador del grupo (aquél con identificador 0 dentro del grupo) se destina a las funciones de *maestro*, gestionando las tareas que enviará al resto de

procesadores, para luego recibir los resultados parciales obtenidos tras la ejecución y combinarlos en la solución final.

El resto de procesadores dentro del grupo desarrollan el rol de *esclavos*, recibiendo del maestro las tareas a ejecutar y los datos necesarios para ello. Tras ejecutar estas tareas, enviarán al procesador maestro los resultados y quedarán a la espera de obtener una nueva tarea del maestro, o bien la señal que les indica que todas las tareas han sido realizadas y el constructo paralelo ha finalizado.

Una primera consideración en cuanto a nuestra implementación de este modelo, es la necesidad de disponer de al menos tres procesadores (un procesador maestro y dos esclavos) para poder realizar tareas en paralelo. Si el grupo sólo dispone de dos procesadores, uno de ello deberá ser el maestro y el otro el esclavo, siendo todas las tareas ejecutadas por este último tal y como se haría de forma secuencial, añadiendo además una sobrecarga con la gestión del modelo y las comunicaciones. Para evitar esto, `llCoMP` comprueba que se disponga al menos de tres procesadores en el grupo actual y, si no es así, el código se serializa para evitar la sobrecarga.

El proceso de generación de código es diferente para el procesador maestro y para los esclavos. Cada patrón contiene ambos códigos para cada una de las etapas en las que se divide la gestión de este constructo: *inicialización*, *gestión de tareas* y *ejecución-comunicación-finalización*. Además, existe un patrón especial para gestionar las *barreras de sincronización*. A continuación se expondrán las características más relevantes de cada uno de estos patrones.

3.5.5.1. Inicialización

Durante la inicialización se almacena la información previa del grupo inicial, para luego restaurarlo al finalizar la ejecución del constructo. Una vez salvado el estado original, se procede a subdividir este grupo de nP procesadores en otros tantos nP grupos uniprocesadores. Durante esta etapa también se reserva la memoria para los mensajes de control que el maestro y los esclavos intercambiarán durante la gestión de las tareas. Este código es común para todos los procesadores.

Procesador maestro: Adicionalmente, el procesador maestro declarará e inicializará el vector de información de los esclavos, que en cada momento le indicará qué tarea está ejecutando cada uno de ellos o si se encuentra en estado de espera. Este vector también contiene una lista de direcciones

de memoria donde el maestro tendrá que almacenar los resultados que le enviarán los esclavos.

3.5.5.2. Gestión de tareas

El código de gestión de tareas sólo afecta al procesador maestro. La gestión se realiza mediante macros, que se autogeneran para cada tarea. La elección de generar este código mediante macros y no usando para ello funciones, se debe a la necesidad de mantenerse dentro del ámbito de las variables implicadas en la ejecución de tareas, debido a que es necesario acceder a las mismas para enviar datos a los esclavos y actualizarlas con las soluciones que se reciban de los mismos. Esta misma aproximación no sería posible con funciones, ya que se perdería el ámbito de estas variables.

De este modo, en el código del procesador maestro se intercalan referencias a estas macros de gestión cada vez que es necesario, y mediante las mismas se indica a los esclavos que ejecuten la tarea en cuestión. El código de estas macros incluye la selección del esclavo que deberá ejecutar la tarea, o bien un estado de espera en el maestro hasta que haya un esclavo libre si en ese momento no lo hay. Una vez seleccionado el esclavo, se actualizará la información del vector y se realizará el envío al de los datos necesarios para que éste ejecute la tarea.

3.5.5.3. Ejecución, comunicación y finalización

Procesador maestro: De estas tres etapas implementadas en el patrón estático, el procesador maestro sólo está afectado por la finalización, ya que el maestro no ejecuta las tareas y las comunicaciones en el maestro se resuelven en la etapa de gestión de tareas, salvo la distribución final de resultados.

Durante la finalización, el procesador maestro deberá asegurarse que todos los esclavos han terminado de ejecutar sus tareas, para entonces enviarles el código de terminación y liberar el vector de información de esclavos. Este comportamiento fuerza un punto de sincronización, momento en el que el maestro distribuye los resultados finales a todos los esclavos, asegurando así la consistencia de memoria.

Procesadores esclavos: Durante la ejecución, los procesadores esclavos se encuentran en un bucle sin fin en el cual pasan por las siguientes etapas:

1. Se encuentran en estado de espera

2. Reciben del maestro el identificador de la tarea a ejecutar y los datos

3. Ejecutan la tarea

4. Envían al maestro los resultados

5. Vuelven al estado de espera hasta recibir una nueva tarea (paso 1) o el código de terminación para abandonar el bucle de espera

Todas las comunicaciones entre el maestro y los esclavos son punto a punto y se realizan mediante llamadas a las funciones `MPI_Send()` y `MPI_Recv()` de `MPI`.

Una consecuencia de la implementación de este modelo es que los esclavos "recuerdan" los valores obtenidos en la última ejecución, es decir, las variables conservan los valores de la de la última tarea ejecutada. Por este motivo, si fuese necesario restaurar alguna variable a los valores iniciales antes de una nueva ejecución, la directiva `task_slave_set_data` proporciona el medio para que el usuario indique estas variables afectadas y sus valores iniciales. Esta inicialización se produce insertando directamente las asignaciones en el código del esclavo, por lo que no involucran comunicación alguna.

3.5.5.4. Barrera

Al usar la directiva `taskq_barrier` de `llc` (subapartado 2.8.2.8, página 92) se fuerza un punto de sincronización. El procesador maestro espera en este punto a que todos los esclavos finalicen sus tareas y comuniquen sus resultados antes de proseguir enviando nuevas tareas.

La implementación de este constructo consiste en detener el envío de nuevas tareas en el maestro hasta que el vector de información de los esclavos indique que todos están en estado de espera, lo que significa que todas las tareas actuales ya han sido finalizadas. El maestro, tras recibir los resultados parciales, continuará la ejecución.

3.5.6. Constructo `barrier`

La traducción es directa, sustituyendo el compilador este constructo por una llamada a la función de `MPI` `MPI_Barrier()` para establecer una barrera de sincronización para el grupo actual.

3.6. Consistencia de memoria en los constructos

Para asegurar la consistencia de la memoria de acuerdo con el modelo OTOSP, los datos modificados deberán ser comunicados durante la operación colectiva. `llc` dispone de varias directivas que cumplen con este fin y que ofrecen diferentes posibilidades según el tipo de acceso a memoria y el constructo en el que son usadas.

La mayoría de estas directivas ya han sido presentadas en apartados previos y estudiado su funcionamiento general. En este apartado se realiza una recopilación de las mismas incidiendo con mayor profundidad en algunos de los detalles de su implementación.

Se pretende que este apartado sirva para una consulta y referencia general más fácil de las mismas, si bien hay que señalar que el funcionamiento específico puede variar ligeramente según el constructo en el que son utilizadas y la topología de las comunicaciones.

3.6.1. `result`: regiones contiguas de memoria

Éste es el caso más sencillo, donde los datos a comunicar comprenden un bloque contiguo de memoria. El usuario especifica en esta directiva un puntero al inicio de la región de memoria y el número de elementos a comunicar. A partir de estos datos el compilador genera el código necesario para realizar la comunicación.

En primer lugar, se calcula el número de bytes a transmitir a través del número de elementos y del tamaño de cada uno de ellos. Estas operaciones están recogidas en los patrones estáticos y, partiendo del puntero especificado por el usuario, se transmiten los bytes calculados. Para ello se utilizan diversas funciones de `MPI`, dependiendo de la topología de las comunicaciones (`MPI_Send()`/`MPI_Recv()`, `MPI_Bcast()`, ...).

En el paradigma de colas de tareas, esta directiva corresponde con `task_master_data` para el envío de datos del maestro a los esclavos, `task_slave_data` para la comunicación de resultados parciales de los esclavos al maestro y `taskq_master_result` para la difusión de los resultados finales por parte del maestro a todos los restantes procesadores (apartado 2.8, página 84).

3.6.2. `rnc_result`: regiones regulares no contiguas de memoria

El siguiente nivel de complejidad a la hora de transmitir los resultados lo encontramos cuando los datos no se encuentran contiguos en memoria, pero se localizan siguiendo un patrón fijo.

Un ejemplo sería una matriz de datos de la que sólo se desea enviar una submatriz. En estos casos, a partir de una posición inicial, el patrón suele estar definido por un número fijo de elementos a enviar y otro número de elementos a omitir, indicando además la cantidad de veces que este patrón se repetirá. Estos parámetros son los que el usuario deberá indicar al usar esta directiva.

A partir de los parámetros especificados el compilador genera el código necesario para las comunicaciones y para empaquetar los datos en un buffer. En este buffer de comunicación se copian los datos de forma contigua y se añade información sobre el patrón de acceso a memoria para que en destino se extraigan los datos y se copien dentro de las variables de forma adecuada.

En el paradigma de colas de tareas la directiva equivalente es `task_slave_rnc_result`, mediante la cual los esclavos envían al maestro datos que siguen un patrón no contiguo uniforme (subapartado 2.8.2.4, página 89).

3.6.3. `nc_result`: regiones no contiguas de memoria

El caso más complejo se produce cuando el acceso a memoria no sigue un patrón regular. A diferencia de los casos anteriores, esta directiva no se coloca al inicio del constructo, sino inmediatamente antes de la sentencia que accede a memoria. De este modo, al realizarse el acceso, queda registrado qué zona de memoria se visitó y qué cantidad de datos fueron modificados.

Una dificultad añadida a esta situación es que el espacio de direcciones difiere de un procesador a otro. Por este motivo las direcciones son calculadas como desplazamientos relativos a una base que se obtiene con la dirección inicial de la variable accedida.

Para gestionar estos registros de accesos, el compilador genera el código que gestiona una lista de las variables afectadas que contiene la dirección base de cada una de ellas. Cada elemento de esta lista de variables dispone, a su vez, de su propia lista de regiones de memoria accedidas, representadas por el desplazamiento a esas zonas y el tamaño del acceso.

A partir de los datos que contiene esta lista, el compilador elabora el código necesario para crear el buffer que contenga todos los datos representados y la información necesaria para que en destino se copien estos datos en las zonas de memoria adecuadas.

Dependiendo del tipo de aplicaciones, la sobrecarga que puede generar la gestión de estas listas puede ser significativa. En muchos de los códigos científicos en los que se ha trabajado se ha constatado que este tipo de accesos se llevan a cabo mayoritariamente cuando se accede a un vector mediante un índice que no sigue un patrón conocido, o bien el acceso depende de alguna condición para efectuarse. En estas ocasiones es muy probable que un gran número de regiones sean consecutivas y puedan ser fusionadas, reduciendo de este modo el número de elementos de las listas con lo que se optimiza su gestión, así como el tamaño de las comunicaciones.

Con el objetivo de compactar aquellas regiones que sean consecutivas o se solapasen, se desarrolló un algoritmo de compactación que aseguraba el mínimo de regiones posibles. Sin embargo, las pruebas realizadas indicaban que para un número relativamente elevado de compactaciones este algoritmo producía una sobrecarga demasiado elevada.

Por este motivo se abandonó el uso de un algoritmo exacto y se incorporó una heurística que obtiene un amplio número de compactaciones añadiendo una baja sobrecarga, tanto en el caso de que se encuentren pocas regiones a compactar, como muchas.

Esta heurística se basa en una inserción ordenada de las regiones según su dirección de comienzo. Al insertar una nueva región se comprueba si puede ser compactada o incluida en una región ya existente. Si es así, sólo habrá que actualizar la información de la región previa, evitando la generación de nuevas regiones. La actualización de la información de una región requiere que se comprueben las regiones colindantes por si hubiera más solapamientos.

Para evitar que el tiempo consumido por la heurística sea excesivo, se han definido una serie de parámetros que controlan el número de intentos y regiones que la heurística tiene en cuenta para realizar las compactaciones. Estos parámetros son modificables en el fichero de cabecera de `llCoMP` y, por defecto, se han establecido a los valores que experimentalmente han demostrado un buen compromiso entre baja sobrecarga y buenos resultados.

3.6.4. `lastresult`: último resultado

La directiva `lastresult` (o `lastprivate` en su versión `OpenMP`) es una variante de la directiva `result` y su implementación es muy similar. La diferencia fundamental reside en que con la directiva `lastresult` se debe incluir la comprobación de que sólo el grupo de procesadores que realizó la última ejecución será el encargado de llevar a cabo el envío y todos los demás grupos recibirán los datos.

3.6.5. `reduce`: reducciones

Este es un caso especial en el cual debe realizarse alguna operación sobre los datos que se comunican. El esquema general de las reducciones se puede resumir en los siguientes pasos:

1. Todos los grupos de procesadores envían sus resultados parciales al procesador maestro.
2. El maestro realiza la operación de reducción sobres estos datos a medida que los va recibiendo.
3. El maestro distribuye el resultado final a todos los restantes procesadores.

`llc` permite que se realicen, además de operaciones escalares, operaciones vectoriales que afectan, elemento a elemento, al rango del vector indicado en la directiva. Por otro lado, también se permite que el usuario defina su propia operación de reducción. La implementación de las reducciones realizada por `llCoMP` depende del constructo en cuestión y, en general, cuenta con un patrón estático de comunicaciones dedicado.

Debido a que `llCoMP` genera código para memoria distribuida, es necesario definir una variable auxiliar para realizar la operación de reducción. Esta variable auxiliar se especifica directamente si se utiliza la sintaxis de `llc`, o bien será el compilador el encargado de definir esta variable si se utiliza la cláusula `reduction` de `OpenMP`, caso en el que debe ir complementada con una directiva `reduction_type` de `llc` que el compilador utiliza para obtener el tipo de las variables afectadas.

Para el paradigma de colas de tareas se utilizan las directivas equivalentes `task_reduce_slave` si se indica la variable auxiliar, o bien `task_t_reduce_slave` si se indica el tipo.

3.6.6. Empaquetado/Desempaquetado

El *empaquetado y desempaquetado de datos* no es una directiva en sí, sino procesos que se han añadido en las directivas para optimizar las comunicaciones. Por este motivo será estudiado en este subapartado.

Estos procesos se activan de forma transparente al usuario cuando se utiliza más de una de las directivas tratadas[5] y/o en una sola directiva se especifican varias regiones de memoria. El objetivo es empaquetar en una misma comunicación los datos de todas las instancias de los distintos comunicadores, añadiendo las cabeceras necesarias en el origen para que en destino puedan extraerse y copiarse en las correspondientes posiciones de memoria.

Para ello se hace uso de los patrones dinámicos, mediante los cuales se genera el código necesario para todo el proceso de empaquetado y envío y, posteriormente, de recepción y desempaquetado. Las funciones de los patrones dinámicos dependen de si se aplican en el origen o el destino.

Origen: El primer paso es la creación del buffer de envío. Para ello es necesario determinar el tamaño del mismo, que se calcula teniendo en cuenta la suma de los tamaños de todos los datos a enviar, junto con las cabeceras que contienen la información que será utilizada en el proceso de desempaquetado.

Una vez determinado el tamaño, se reserva la memoria y se realiza el proceso de empaquetado, en el cual se copia en el buffer los datos a comunicar, intercalando la información de empaquetado. Esta información depende del constructo en el que se use la directiva y de la topología, conteniendo en general datos para identificar qué variables y zonas de memoria se envían, tamaño de cada paquete, etc.

Destino: Para reducir el número de comunicaciones, en el destino se calculará el tamaño del buffer de recepción necesario siempre que sea posible, por lo cual la transmisión se puede realizar con sólo un paquete. Cuando esto no es posible, se llevarán a cabo dos comunicaciones, una primera de tamaño fijo que indica el tamaño del buffer de recepción y otra segunda que contiene los datos.

Una vez obtenido el tamaño de la comunicación, se reserva el espacio para el buffer y se procede a la recepción de los datos. Acto seguido, se lleva a cabo el proceso de desempaquetado, para lo cual se utilizan

[5] excepto `nc_result` que no puede ser gestionada en tiempo de compilación

las cabeceras que contienen la información sobre en qué variables, en qué posición y cuántos datos se deben copiar.

3.7. Traducción de funciones OpenMP

Las funciones de OpenMP implementadas en llc devuelven información acerca del número de procesadores del grupo actual, el identificador del procesador dentro del grupo, etc. llCoMP almacena esta información en unas variables predefinidas, por lo que la traducción consiste en la sustitución de la llamada a la función por la correspondiente variable. Este proceso se realiza durante la generación de código que se produce en la etapa de análisis sintáctico.

La equivalencia entre funciones de OpenMP y variables de llc usada en el proceso de traducción, puede consultarse en la tabla 3.3. Para cada función se indica cuál es el significado en OpenMP y cuál es la equivalencia en llc.

Función y significado OpenMP	**Variable y significado** llc
omp_get_thread_num() Identificador del thread dentro del equipo.	LLC_NAME Identificador del procesador dentro del grupo actual.
omp_get_num_threads() Número total de *threads* del equipo.	LLC_NUMPROCESSORS Número total de procesadores dentro del grupo actual.
omp_get_num_procs() Número total de procesadores.	LLC_NUMPROCESSORS Número total de procesadores dentro del grupo actual.
omp_get_global_thread_num() Ninguno. Función añadida en llc.	LLC_GLOBAL_NAME Identificador del procesador dentro del grupo global.
omp_get_global_num_threads() Ninguno. Función añadida en llc.	LLC_GLOBAL_NUMPROCESSORS Número total de procesadores dentro del grupo global.

Tabla 3.3: Funciones OpenMP con traducción en llc.

Si el usuario lo desea, puede utilizar las variables llc mencionadas directamente en el código al programar usando llc. Se debe tener en cuenta que el acceso a estas variables debe ser únicamente de lectura y que

modificar el valor de las mismas puede conllevar una ejecución errónea de la aplicación. En cualquier caso, no se aconseja su uso directo para no perder la compatibilidad con el programa secuencial.

3.8. Traducción de macros llc

Las macros de llc no se traducen, sino que durante el proceso de generación de código se incluye el fichero de cabecera llc.h que contiene la expansión de estas macros. La lista completa de macros de llc está recogida en la tabla 2.7 de la página 103 .

En la tabla 2.7 además se indica el tipo de operación: de entrada, salida o tratamiento de ficheros. A modo de ejemplo, en el listado 3.4 se presenta la expansión de una macro de cada uno de estos tipos de operación. En concreto la macro de operación de entrada LLC_fread (línea 1), de salida LLC_fwrite (línea 9) y de tratamiento de ficheros LLC_fclose (línea 11).

```
1  #define LLC_fread(ptr,size,items,fd) {          \
2    if (LLC_GLOBAL_NAME == 0) {                   \
3      fread(ptr, size, items, fd);                \
4    }                                             \
5    MPI_Bcast (ptr, (size * items), MPI_BYTE,     \
6                        0, *llc_CurrentGroup);    \
7  }

9  #define LLC_fwrite      if (LLC_GLOBAL_NAME == 0) fwrite

11 #define LLC_fclose      if (LLC_GLOBAL_NAME == 0) fclose
```

Listado 3.4: Ejemplos de expansión de macros llc

3.9. Otras traducciones de llCoMP

Además de las traducciones de los constructos y directivas, funciones y macros, durante el proceso de compilación se llevan a cabo otros procesos de traducción que describiremos a continuación.

3.9.1. Inclusión de fichero de cabecera de llc

La primera acción que llCoMP realiza para generar el código de salida es la inclusión del fichero de cabecera llc.h en la primera línea, mediante la sentencia #include ‘‘llc.h’’.

Dentro de este fichero de cabecera se encuentran declaraciones de tipos de datos, definiciones de constantes, expansiones de macros y declaración de algunas variables globales en los códigos de llc, como comunicadores, variables que controlan el número e identificador de los procesadores dentro de los grupos locales y globales, etc.

Esta cabecera también contiene la inclusión del fichero de cabecera de MPI (mpi.h). Este fichero es necesario para la posterior compilación y enlace con las librerías de MPI en el código de salida generado.

3.9.2. Traducción de la función exit

El código secuencial de entrada puede contener llamadas a la función exit de C. Esta situación debe evitarse, puesto que si alguno de los procesadores alcanza esta sentencia finalizará la ejecución sin realizar una salida limpia del entorno paralelo, lo que provocará una ejecución errónea de la aplicación.

Para evitar esto, durante el análisis sintáctico se realiza la traducción de la función exit de C por la macro LLC_EXIT de llc. Esta macro incluye una llamada a la función MPI_Abort() de MPI para proporcionar una salida limpia al entorno paralelo creado en tiempo de ejecución por MPI. La expansión de esta macro está recogida en el fichero de cabecera llc.h y se presenta en el listado 3.5.

```
1 #define LLC_EXIT(code) {                    \
2   MPI_Abort (MPI_COMM_WORLD, code);         \
3   exit(code);                               \
4 }
```

Listado 3.5: Definición de la macro LLC_EXIT

3.9.3. Traducción de la función main

Para la correcta ejecución en paralelo del código generado por llCoMP es necesario realizar una serie de operaciones que no están presentes en el

código secuencial original, como inicializar el entorno paralelo, guardar en variables el número de procesadores del grupo y el identificador dentro del mismo, finalizar el entorno paralelo después de la ejecución, etc.

Estas operaciones deben ser ejecutadas al comienzo de la aplicación para asegurar el correcto funcionamiento del código paralelo generado. El mejor punto para llevarlas a cabo es en la propia función `main()`. Sin embargo, la introducción de código en la función `main()` original puede crear conflictos.

Para solucionar este problema, se renombra la función `main()` del código original por `llc_main()`, y `llCoMP` introduce una función `main()` autogenerada que contiene el código necesario para gestionar el entorno paralelo.

El contenido de esta función `main` se recoge en el listado 3.6. En este listado se puede observar cómo una vez inicializado el entorno de ejecución de `llc` se realiza en la línea 14 una llamada a `llc_main()` (función `main()` original).

```
1  /* LLC Autogenerated main */

3  int main (int argc, char **argv) {
4    int llc_return_code;

6    MPI_Init(&argc, &argv);
7    MPI_Comm_rank(MPI_COMM_WORLD, (int*)&LLC_GLOBAL_NAME);
8    MPI_Comm_size(MPI_COMM_WORLD, (int*)&
       LLC_GLOBAL_NUMPROCESSORS);
9    LLC_NAME = LLC_GLOBAL_NAME;
10   LLC_NUMPROCESSORS = LLC_GLOBAL_NUMPROCESSORS;
11   llc_GlobalGroup = MPI_COMM_WORLD;
12   llc_CurrentGroup = &llc_GlobalGroup;

14   llc_return_code = llc_main (argc, argv);

16   MPI_Finalize();
17   return llc_return_code;
18 }
```

Listado 3.6: Traducción de la función `main`

Capítulo 4

Algoritmos y resultados computacionales

4.1. Introducción

Este capítulo recopila el trabajo experimental que hemos desarrollado para validar nuestra aproximación. Es importante reseñar que no presentaremos aquí ni todos los algoritmos que han sido desarrollados durante la realización de la tesis, ni todos los resultados computacionales que han sido obtenidos con ellos. Hemos tratado de recoger aquellos códigos que son más relevantes y un resumen ilustrativo de los resultados que se han obtenido. Por otra parte, y en cuanto a los resultados computacionales, destacar también que no nos hemos limitado a presentar los resultados más satisfactorios, sino que mostramos también situaciones en las que nuestra aproximación no obtiene un alto rendimiento.

En la última etapa de realización de este trabajo nos hemos esforzado en validar la aproximación de `llc` con casos de utilidad práctica real. Muestra de ello son tanto la colaboración (a través del proyecto *Comparhe* [3]) con el Grupo de Computación Científica Paralela (PSCOM) [129] de la Universidad Jaime I de Castellón (UJI) [147] (véanse los apartados 4.5.4 y 4.5.5), como el trabajo desarrollado en colaboración con investigadores del IAC [87].

Uno de los objetivos que hemos tenido presentes a la hora de diseñar nuestro lenguaje y su compilador ha sido la eficiencia del código generado. Mediremos esta eficiencia comparando siempre que sea posible los resultados computacionales de nuestra aproximación con aquellos obtenidos con los lenguajes que usamos como referencia, `MPI` y `OpenMP`. Nuestra meta es que

el código `MPI` que genera `llCoMP` obtenga unos resultados comparables a la mejor versión *ad hoc*.

Como criterio de medida de rendimiento de nuestras aplicaciones hemos elegido la comparación del tiempo de ejecución consumido por las regiones de código paralelizadas frente al tiempo secuencial de las mismas. Para la toma de tiempos hemos utilizado una serie de macros que se basan en el uso de la función `gettimeofday()`. En general, en este capítulo presentaremos los resultados mediante gráficas que muestran la aceleración obtenida frente al número de procesadores utilizados, si bien en la mayoría de casos presentamos también tablas de tiempos para que el lector pueda tener una referencia del orden de los tiempos obtenidos.

En la toma de resultados computacionales hemos tratado de buscar la mayor heterogeneidad en los equipos utilizados, así como en las condiciones de ejecución. En este capítulo hemos llevado a cabo una recopilación de los algoritmos que hemos considerado más interesantes y representativos de aplicaciones paralelizadas con `llc`.

Hemos clasificado los diferentes algoritmos en cuatro grandes grupos, acordes con el paradigma de paralelización usado. La mayoría de códigos presentados se han extraído de aplicaciones científicas reales que se usan comúnmente en campos tales como ingeniería, física, astrofísica, matemáticas, etc. Hemos diseñado este capítulo catalogando los ejemplos desde más simples a más complejos, de forma que pueda utilizarse como un tutorial que indique al usuario cómo realizar la paralelización de códigos mediante `llc`. Es posible obtener el código completo de estas aplicaciones en [100].

La descripción de las directivas de `llc` que se hace en este capítulo tiene un carácter marcadamente práctico. Para consultar la sintaxis o semántica de cada una de estas directivas, el lector puede referirse al capítulo 2 dedicado a la descripción del lenguaje `llc`.

En muchos de los ejemplos que presentaremos en este capítulo hemos partido de una paralelización previa realizada en `OpenMP`. A pesar de que en todos los casos pueden ser paralelizados usando exclusivamente directivas de `llc`, hemos decidido conservar la sintaxis de `OpenMP` para comparar allí donde sea posible los resultados computacionales obtenidos con ambos lenguajes usando el mismo código fuente. Por otro lado, también perseguimos el objetivo de mostrar el grado de compatibilidad que existe entre ambos lenguajes y la simplicidad con la que se pueden obtener códigos `llc` a partir del equivalente `OpenMP`, si bien en algunas ocasiones la paralelización directa usando únicamente directivas `llc` puede resultar más sencilla.

El apéndice A recopila la descripción de los sistemas computacionales utilizados en la toma de resultados que se referencian en este capítulo. La tabla A.3 de la página 260 contiene un resumen de las características principales de estos equipos.

4.2. Paralelización de bucles

En este apartado realizaremos un recorrido por varios ejemplos `llc` que ilustran el uso de las directivas de `llc` para la paralelización de bucles en distintos contextos. Con respecto a otros paradigmas de paralelización de `llc`, los bucles constituyen uno de los métodos más usados para obtener códigos paralelos, razón por la cual este apartado tiene un mayor peso y cuenta con un amplio conjunto de algoritmos.

4.2.1. Algoritmo del `Conjugado del Gradiente`

El listado 4.1 recoge un bucle de inicialización que pertenece al código del conjugado del gradiente (`CG`) [76, 99]. Este algoritmo usa el método de las potencias inversas para encontrar una estimación del mayor autovalor de una matriz dispersa definida positiva con una distribución aleatoria de elementos no nulos.

El código de partida al que hemos aplicado la paralelización `llc` ha sido extraído de la implementación en `OpenMP` de los *NAS Parallel Benchmarks* (NPB) [112, 9, 89], un conjunto de aplicaciones paralelas que ha sido desarrollado por la NASA Advanced Supercomputing (*NAS*) Division [113] con el fin de evaluar el rendimiento de supercomputadoras masivamente paralelas. La implementación en `OpenMP` de partida ha sido realizada en el marco del proyecto `Omni` [120].

```
1  for(i = 0; i < cols; i++) {
2    p[i] = r[i] + beta*p[i];
3  }
```

Listado 4.1: Bucle de inicialización en el Conjugado del Gradiente

El bucle del listado 4.1 inicializa el vector `p`, siendo sus iteraciones independientes unas de otras, por lo que este bucle nos sirve para ejemplificar

la paralelización con `llc` de un bucle simple sin dependencias. El resultado de esta paralelización se muestra en el listado 4.2.

```
1 #pragma omp parallel for
2 #pragma llc result (&p[i], 1)
3   for(i = 0; i < cols; i++)  {
4     p[i] = r[i] + beta*p[i];
5   }
```

Listado 4.2: Bucle paralelizado en el Conjugado del Gradiente

La directiva `result` de la línea 2 se utiliza para garantizar la consistencia de memoria entre los diferentes grupos de procesadores que han ejecutado el bucle. Nótese que sólo es necesario indicar un puntero al principio de la región y el número de elementos a comunicar, sin que sea necesario especificar el tipo de los mismos, tal y como sí se requiere en otros lenguajes. El algoritmo `CG` contiene varios bucles similares al que hemos presentado, siendo el código mostrado uno de los más simples.

La figura 4.1 presenta la aceleración del algoritmo `CG` en el `Cray T3E` hasta un total de 34 procesadores para un tamaño de problema medio (`MID`) con 1853104 elementos distintos de cero, tamaño elegido entre los tres disponibles (`TINY`, `MID`, `LARGE`) debido a que el tamaño mayor presentaba problemas de memoria en algunas máquinas. Los resultados obtenidos muestran una aceleración logarítmica. Este comportamiento viene justificado por la poca granularidad de los bucles paralelizados, que en su mayoría contienen simples asignaciones, con una complejidad similar al bucle mostrado en el listado 4.2. De este modo, a partir de un cierto número de procesadores se alcanza el techo del valor de la aceleración, por lo que añadir más procesadores no produce ninguna mejora en el rendimiento.

En este tipo de aplicaciones de grano muy fino, `OpenMP` suele presentar un rendimiento considerablemente mejor al de alternativas de memoria distribuida, como `llc` o `MPI`. Esto se debe a que para cómputos poco intensivos, la ganancia que se obtiene al efectuarlos en paralelo puede verse igualada o incluso superada por el costo de las posteriores comunicaciones necesarias en los entornos distribuidos para transmitir los resultados.

Para ilustrar este comportamiento, la figura 4.2 recoge los resultados obtenidos con el algoritmo `CG` en la `SunFire 6800`, donde es posible ejecutar tanto la versión `llc` como la de `OpenMP`. La figura 4.2 muestra una aceleración superior de `OpenMP` frente a `llc`, prácticamente del doble, observándose, además, que la curva de `OpenMP` está afectada en menor medida por la

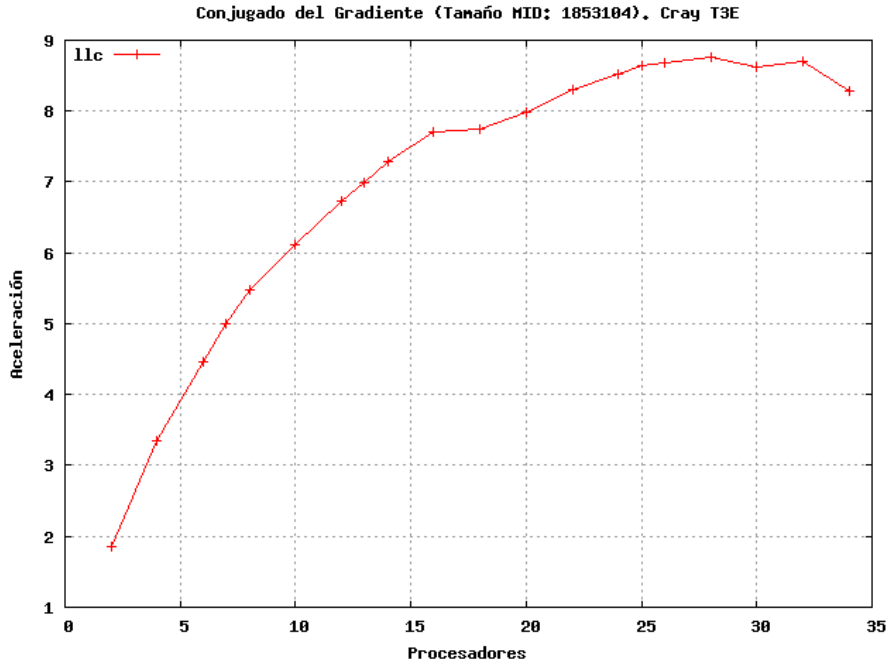

Figura 4.1: Aceleración para el código `CG` con `llc` en el `Cray T3E`

limitación de la aceleración, que en `llc` provoca una curva logarítmica prematura. A este resultado contribuye el hecho de que, después de un estudio de rendimiento, se consideró oportuno no paralelizar todos los bucles posibles en `llc`, por lo que la versión de `OpenMP` cuenta con más bucles paralelizados.

La tabla 4.1 recoge los tiempos –medidos en segundos– utilizados para la confección de las figuras 4.1 y 4.2, en las máquinas `Cray T3E` y `SunFire 6800`, respectivamente, para un tamaño de problema medio (N = 1853104). Los resultados no se muestran para todos los procesadores del rango ya que generalmente en aquellas máquinas con más de una decena de procesadores solemos lanzar ejecuciones cada dos procesadores (normalmente para procesadores pares), con el fin de economizar el tiempo de ejecución disponible, si bien también añadimos algunos procesadores impares como medida de comprobación.

	Cray T3E	SunFire 6800	
procs.	`llc`	`llc`	`OpenMP`
SEQ	148.58	53.3	53.3
2	79.7947	26.7825	22.4598
4	44.4879	15.8941	12.0345
6	33.2504	12.2495	8.34363
7	29.6829		
8	27.111	10.2954	6.87655
9		9.65228	6.14221
10	24.3005	9.22732	5.69491
12	22.067	8.47776	4.78993
13	21.2693		
14	20.4069	8.21537	4.41688
16	19.259	7.92908	3.95606
17		7.51476	3.79589
18	19.1972	7.82508	3.80336
20	18.5909	7.59366	3.66638
22	17.8952	7.48857	3.68462
24	17.4447		
25	17.1808		
26	17.1333		
28	16.9477		
30	17.2173		
32	17.0599		
34	17.9156		

Tabla 4.1: Tiempos (en segundos) obtenidos para el código `CG`

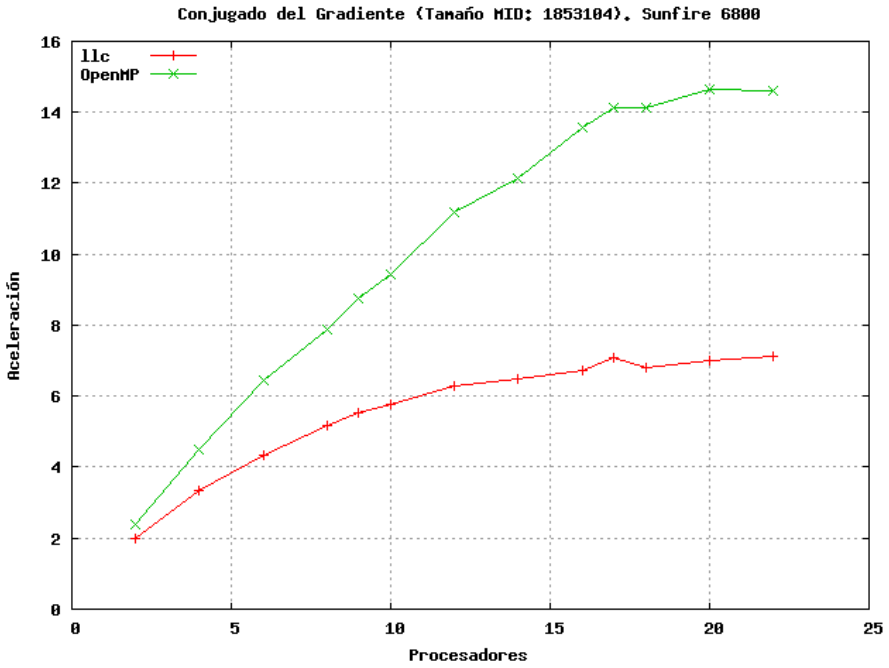

Figura 4.2: Aceleración para el código `CG` con `llc` y `OpenMP` en `SunFire 6800`

4.2.2. Algoritmo `Multibaseline Stereo`

El código `Multibaseline Stereo` implementa un método de visión binocular estéreo usado para obtener información tridimensional de una escena. La estereovisión es computacionalmente intensiva y la naturaleza espacialmente repetitiva de este método favorece su paralelización.

La principal desventaja de este algoritmo es el problema de la correspondencia con el punto de la imagen. El compromiso entre la precisión (a la que ayuda la separación entre cámaras) y la facilidad de realizar la correspondencia ha sido obtenida usando múltiples cámaras u ubicaciones, lo que se conoce como *Multibaseline Stereo*. Este algoritmo [119] ha sido extraído de la *CMU task parallel suite* [47] y consiste en una adaptación de una implementación previa de paralelismo de datos [153] que ofrece una mejor precisión a través del uso de tres cámaras.

El listado 4.3 ilustra una situación donde se realiza un acceso a una región de memoria de una estructura (líneas 10 y 11), en concreto una matriz direccionada a través de un puntero. La directiva `result` de la línea 7 sólo necesita recibir el puntero al inicio a la región contigua de memoria que se va a comunicar (macro `DIFFIMG` definida en la línea 1) y el número de elementos.

```
1  #define DIFFIMG(i, j)    diffimg->data[i][j]
2  #define REF(i, j)        ref->data[i][j]
3  #define M1(i, j)         m1->data[i][j]
4  #define M2(i, j)         m2->data[i][j]

6  #pragma omp parallel for
7  #pragma llc result (&DIFFIMG(i,0), (COLS - 2 * curdisp))
8    for (i = 0; i < ROWS + WINY; i++)
9      for (j = 0; j < COLS - 2 * curdisp; j++) {
10       DIFFIMG(i, j) = SQR(REF(i, j) - M1(i, j + curdisp)) +
11           SQR(REF(i, j) - M2(i, j + 2 * curdisp));
12     }
13 }
```

Listado 4.3: Bucle paralelizado en el algoritmo `stereo`

La figura 4.3 muestra los resultados computacionales obtenidos en el `Cluster UJI-SPINE` [134]. El código utilizado ejecuta en un bucle `for` externo que se aplica un número dado de iteraciones (`ITER`), tomándose como valor de referencia la media de los tiempos obtenidos. En este caso se ha

trabajado con una imagen de tamaño 500 × 500, con 16 y 32 iteraciones, observándose cómo hasta 8 procesadores los resultados de ambas ejecuciones son similares, mostrando un comportamiento cercano a la linealidad. A partir de 8 procesadores la aceleración se mantiene prácticamente constante para la ejecución de 16 iteraciones.

Como en otros casos, al trabajar con un número de procesadores cercano al máximo que la configuración de la máquina nos ofrece, se corre el riesgo de que alguno de ellos sea desasignado. Para mitigar este efecto, hemos repetido las ejecuciones con 32 iteraciones, por lo que los resultados se ven menos afectados y la aceleración obtenida aumenta.

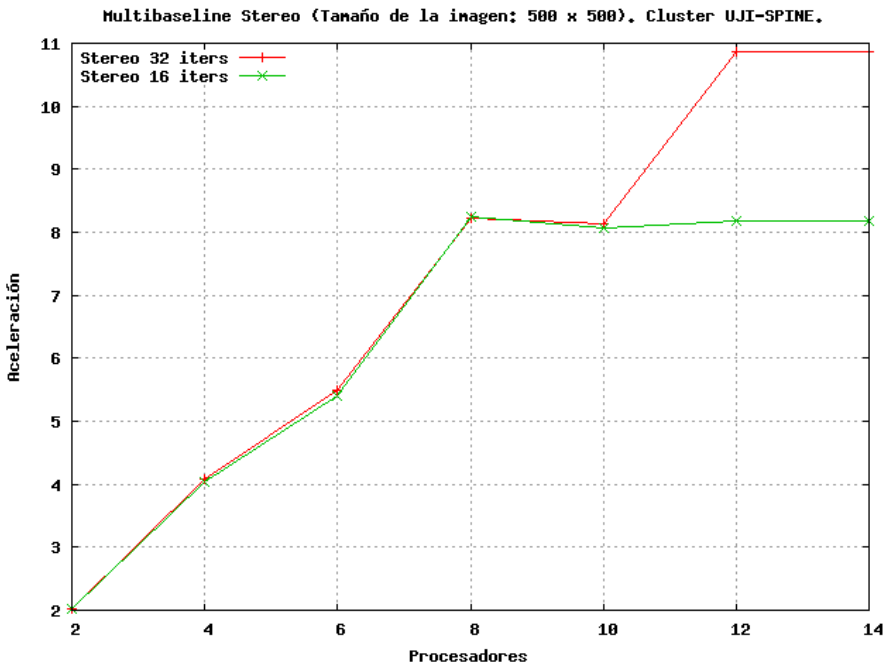

Figura 4.3: Aceleración para el código `stereo` con `llc` en el `Cluster UJI-SPINE`

La figura 4.4 se obtuvo en una de las divisiones SMP del `Cluster EPCC` [134], lo que permite comparar los resultados obtenidos por `llc` con los de `OpenMP`. Como se desprende de esta figura, la versión `llc` tiene un rendimiento semejante al de `OpenMP`, siendo incluso ligeramente superior, si bien no se puede establecer una comparación directa entre `llc` y `OpenMP` puesto que explotan modelos diferentes, influyendo también el hecho de que las aproximaciones de memoria compartida suelen presentar peor escalabilidad por problemas del ancho de banda de acceso.

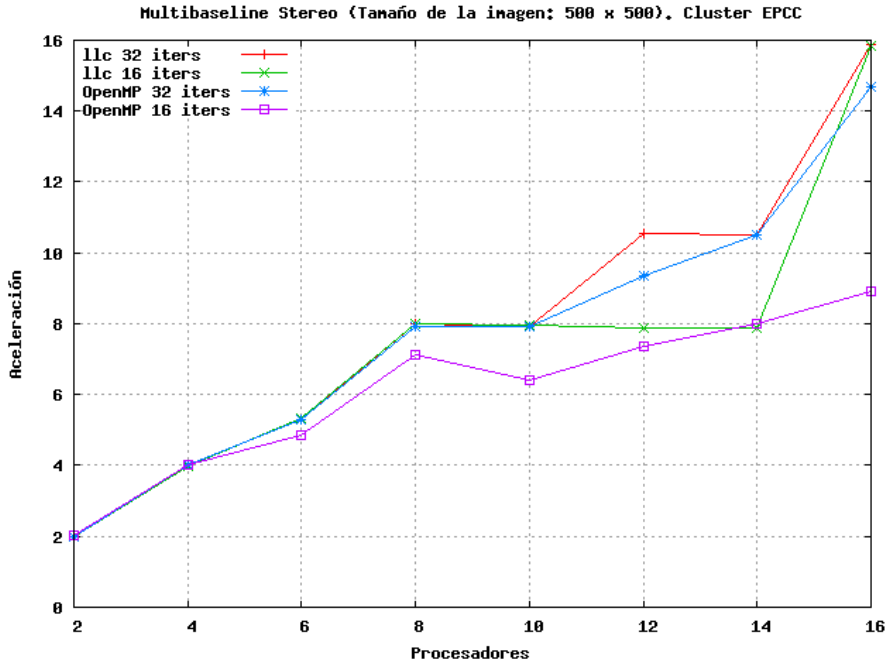

Figura 4.4: Aceleración para el código `stereo` con `llc` y `OpenMP` en el `Cluster EPCC`

La tabla 4.2 muestra los tiempos a partir de los cuales se han obtenido las aceleraciones presentadas en las figuras 4.4 y 4.3, para 16 y 32 iteraciones con las versiones de `llc` y `OpenMP`, tanto en el `Cluster UJI-SPINE` como en el `Cluster EPCC`, en ambos casos utilizando una imagen de tamaño 500×500.

	Cluster UJI-SPINE		Cluster EPCC			
	`llc`		`llc`		`OpenMP`	
procs.	16 it.	32 it.	16 it.	32 it.	16 it.	32 it.
SEQ	3.06023	6.18894	11.9671	23.96435	11.40688	22.84992
2	1.51149	3.05007	5.92644	11.9737	5.64066	11.3482
4	0.75663	1.51821	2.97743	5.96356	2.83744	5.69964
6	0.56409	1.14752	2.24792	4.55092	2.34590	4.29848
8	0.37140	0.75645	1.49143	2.99749	1.60198	2.87845
10	0.37822	0.75946	1.50903	3.03315	1.78468	2.87908
12	0.37293	0.56411	1.51651	2.27108	1.55055	2.43825
14	0.37768	0.56334	1.51167	2.28065	1.42929	2.17072
16			0.75619	1.50433	1.28079	1.55238

Tabla 4.2: Tiempos (en segundos) obtenidos para el código `stereo`

4.2.3. Algoritmo de `Dinámica Molecular`

Esta aplicación es una implementación del algoritmo Verlet de velocidad [143, 146] para simulación de dinámica molecular. A partir de las posiciones, masas y velocidades de `np` partículas, el algoritmo calcula la energía del sistema y las fuerzas sobre cada partícula, mediante el uso de un procedimiento numérico iterativo. La precisión de la aproximación obtenida viene determinada por el número de pasos de la simulación, siendo este parámetro, junto con el número de partículas (`np`), los datos de entrada del algoritmo. La paralelización realizada con `llc` tiene como punto inicial el código de `OpenMP` que puede descargarse de su página oficial [122].

```
1  #pragma omp parallel for default(shared) private(i,j)
       firstprivate(rmass, dt)
2  #pragma llc result (&pos[i], 1, &vel[i][0], nd)
3  #pragma llc result (&a[i], 1)
4    for (i = 0; i < np; i++) {
5      for (j = 0; j < nd; j++) {
6        pos[i][j]=pos[i][j]+vel[i][j]*dt+0.5*dt*dt*a[i][j];
7        vel[i][j]=vel[i][j]+0.5*dt*(f[i][j]*rmass+a[i][j]);
8        a[i][j] = f[i][j]*rmass;
9      }
10   }
```

Listado 4.4: Algoritmo de Dinámica Molecular paralelizado mediante un código `llc`

El listado 4.4 recoge uno de los bucles paralelizados del algoritmo de dinámica molecular. En concreto, entre las líneas 6 y 8 se calcula la posición (`pos`), velocidad (`vel`) y aceleración (`a`) para cada una de de las partículas (`np`) y dimensiones (`nd`), que se recorren en los bucles de las líneas 4 y 5, respectivamente. Nótese que los vectores de posición, velocidad y aceleración de cada partícula son, en realidad, matrices, donde el segundo índice especifica la dimensión de la coordenada.

La directiva `result` de la línea 2 recoge dos formas de especificar que, para cada iteración, se deben comunicar todas las dimensiones localizadas en la posición `i` del respectivo vector. Por un lado, para el vector de posiciones `pos` se ha optado por indicar un único elemento a partir del puntero que referencia la primera dimensión de la matriz (`&pos[i], 1`), mientras que para la velocidad se especifica un puntero al primer elemento del vector y se indica que se deben copiar `nd` elementos (`&vel[i][0], nd`). Ambas opciones son

equivalentes y producirán el mismo código `MPI` tras compilarse con `llCoMP`.

La figura 4.5 compara la ejecución de la versión `llc` del algoritmo en el `Cluster UJI-RA` con una versión *ad hoc* en `MPI` [60], habiéndose computado 10 pasos del algoritmo para 8192 partículas de tres dimensiones. En esta figura `MPI` obtiene un mejor comportamiento, con una aceleración casi lineal, mientras que `llc` tiene una tendencia similar, pero con rendimiento inferior.

Achacamos este menor rendimiento al diferente grado de optimización con el que las funciones de comunicación de `MPI` están implementadas en determinadas máquinas. En general, las versiones `llc` y *ad hoc* en `MPI` no hacen uso de las mismas funciones de comunicación, sino que estas últimas utilizan las funciones que mejor se ajustan a las características del problema, mientras que `llc` usa funciones más genéricas para abarcar un mayor conjunto de situaciones.

En otras máquinas, como la `SunFire 6800`, los resultados para ambas versiones `llc` y *ad hoc* `MPI` son prácticamente idénticos, tal y como muestra la figura 4.6 en la que se evalúa el mismo código con los mismos parámetros. Los tiempos tomados para obtener esta figura se recogen en la tabla 4.3.

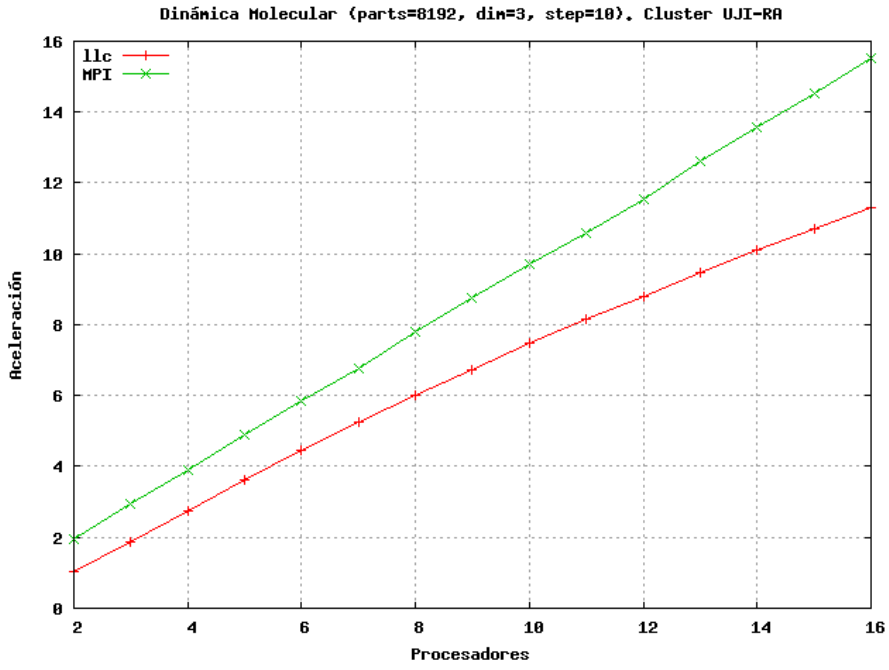

Figura 4.5: Aceleración para el código Dinámica Molecular en el `Cluster UJI-RA`

La figura 4.6 añade los resultados obtenidos para una versión de

OpenMP, si bien el rendimiento es inferior al de llc y MPI. Atribuimos este comportamiento a que la versión OpenMP se ve más afectada por problemas de ancho de banda característicos de memoria compartida, como el cuello de botella producido por las sincronizaciones introducidas al acceder continuamente a las mismas posiciones de memoria, lo cual sucede en menor medida en memoria distribuida ya que se opera sobre copias locales e independientes y sólo se comunican los resultados al finalizar el bucle paralelo.

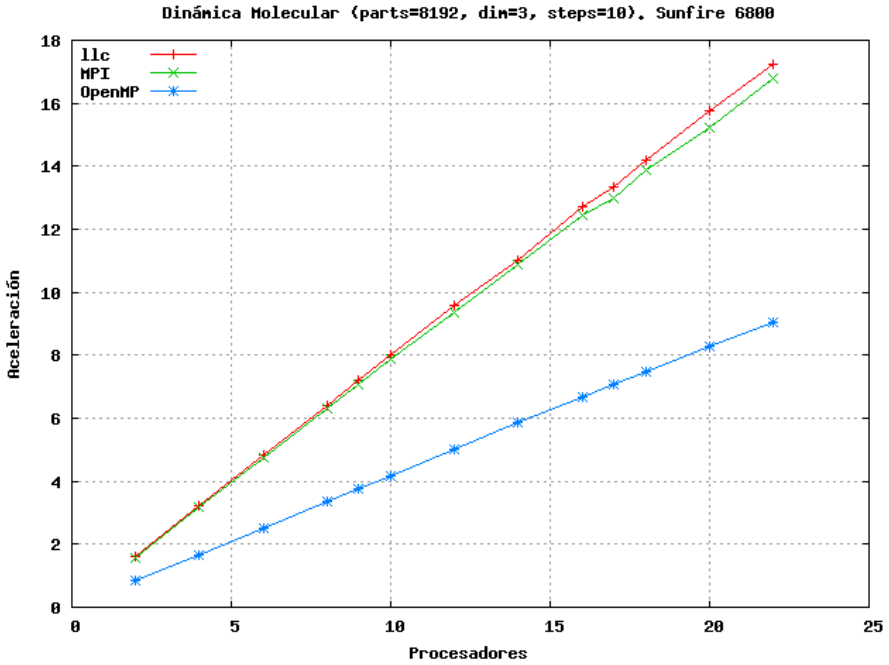

Figura 4.6: Aceleración para el código Dinámica Molecular en el SunFire 6800

4.2.4. Algoritmo FFT de ‘‘6 pasos’’ de Bailey

La *transformada discreta de Fourier (DFT)* es una herramienta importante con múltiples aplicaciones en diversos campos de la ciencia y la ingeniería. La DFT es un producto matriz-vector $y = F_n x$ donde x e y son vectores complejos y $F_n = (f_{pq})$ es una matriz $n \times n$ tal que $f_{pq} = w_n^{pq}$, con $w_n = cos(2\pi/n) - i\ sin(2\pi/n) = e^{-2\pi i/n}$ e $i = \sqrt{-1}$.

La transformada rápida de Fourier es un algoritmo eficiente que saca partido a la estructura de F_n para realizar el producto de la DFT con tiempo $O(n \log n)$. También se pueden definir FFTs de mayores dimensiones,

	SunFire 6800		
procs.	`llc`	`MPI`	`OpenMP`
SEQ	148.6		
2	91.9866	93.8804	177.788
4	46.1022	46.9922	88.9457
6	30.8186	31.3933	59.3018
8	23.1533	23.5491	44.4896
9	20.5971	20.985	39.5333
10	18.5655	18.8937	35.6029
12	15.5182	15.8892	29.672
14	13.4666	13.6303	25.4279
16	11.6868	11.9522	22.2748
17	11.1251	11.452	21.0095
18	10.4846	10.7103	19.8277
20	9.43916	9.75181	17.9495
22	8.62964	8.83949	16.4214

Tabla 4.3: Tiempos (en segundos) obtenidos para el código `Dinámica Molecular`

pudiendo consultar el lector en [101] descripciones más detalladas de diferentes algoritmos.

El algoritmo FFT unidemensional (`1D FFT`) que aquí se presenta ha sido extraído del conjunto de códigos de CMU [47] y corresponde con el algoritmo de *"6 pasos" de Bailey* [8]. Si consideramos el tamaño de la señal de entrada como el producto de dos números, $n_1 \times n_2$, entonces la `1D FFT` puede ser calculada usando `1D FFT`s independientes más pequeñas. El vector de entrada puede ser reorganizado como una matriz $n_1 \times n_2$. Los seis pasos de este algoritmo se resumen a continuación, siendo la matriz resultante de un paso los datos de entrada para el siguiente:

1. Trasponer el conjunto de datos de entrada
2. Realizar n_1 `1D FFT` individuales de n_2 puntos sobre la matriz resultante
3. Multiplicar la matriz resultado A_{ij} por $e^{\pm 2\pi ijk/n}$
4. Trasponer la matriz
5. Realizar n_2 `1D FFT` individuales de n_1 puntos sobre la matriz resultante
6. Volver a trasponer la matriz

```
1  int tpose(complex *a, complex *b, const int n) {
2    register int i, j;
3
4  #pragma omp parallel for private(i, j)
5  #pragma llc result (&b[i * n], n)
6    for (i = 0; i < n; ++i) {
7      for (j = i; j < n; ++j) {
8        b[i * n + j] = a[j * n + i];
9        b[j * n + i] = a[i * n + j];
10     }
11   }
12   return 0;
13 }
14
15 int scale(complex *a, complex *v, const int n) {
16   register int i, j, index;
17   complex aa, vv;
18
19 #pragma omp parallel for private(i, j, index, aa, vv)
20 #pragma llc result (&a[i * n], n)
21   for (i = 0; i < n; ++i) {
22     for (j = 0; j < n; ++j) {
23       index = i * n + j;
24       aa = a[index];
25       vv = v[index];
26       a[index].re = aa.re * vv.re - aa.im * vv.im;
27       a[index].im = aa.re * vv.im + aa.im * vv.re;
28     }
29   }
30   return 0;
31 }
32
33 int cffts(complex *a, int *brt, complex *w,
34           const int n, int logn, int ndv2) {
35   register int i;
36
37 #pragma omp parallel for private(i)
38 #pragma llc result (&a[i * n], n)
39   for (i = 0; i < n; ++i)
40     fft(&a[i * n], brt, w, n, logn, ndv2);
41   return 0;
42 }
```

Listado 4.5: Código de la `FFT` de "6 pasos" de Bailey

El listado 4.5 recoge la paralelización llevada a cabo en `llc` de las operaciones realizadas en estos 6 pasos: cálculo de matriz traspuesta (pasos 1, 4 y 6, función `tpose()` de la línea 1), multiplicación (paso 3, función `scale()`, línea 15) y cálculo de la FFT unidemensional (pasos 2 y 5, función `cffts()` en línea 33). El algoritmo de cálculo de la FFT de `CMU` que hemos utilizado realiza iterativamente un cierto número (`ITERS`) de estos 6 pasos en un conjunto de datos de entrada generados sintéticamente, siendo este número de iteraciones y el tamaño de la matriz ($N = n_1 = n_2$) los parámetros de entrada.

La figura 4.7 presenta los resultados obtenidos para la versión `llc` del código de la `FFT` [134] en 6 pasos en el `Cluster UJI-SPINE` para $N = 2^{20}$ flotantes con 16 y 32 iteraciones. Estos resultados recuerdan a los ya estudiados para otro de los códigos de la *CMU task parallel suite*, concretamente el algoritmo `stereo` (figura 4.4 en la página 158). De igual modo, podemos observar cómo las curvas de la aceleración tienen una tendencia lineal y que al incrementar el número de iteraciones de 16 a 32 los resultados obtenidos muestran un mejor comportamiento.

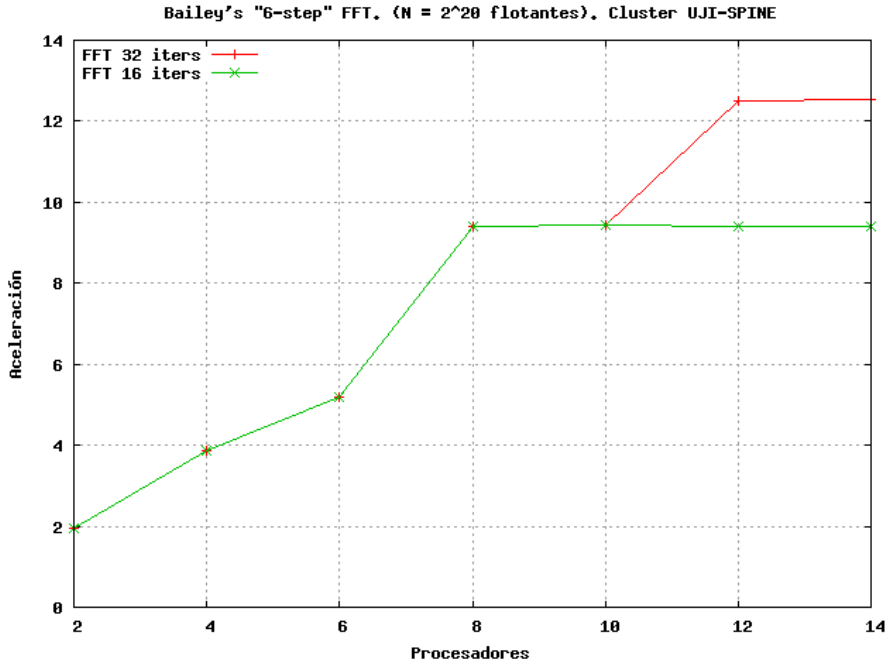

Figura 4.7: Aceleración para el código `FFT` de "6 pasos" de Bailey en el `Cluster UJI-SPINE`

La figura 4.8 recoge los resultados [134] obtenidos en el `Cluster EPCC` y

compara los resultados de `llc` con `OpenMP`. También aquí constatamos cómo `llc` obtiene un mejor comportamiento que `OpenMP`, siendo la diferencia entre ambas curvas mayor al aumentar el número de procesadores. Como en otras ocasiones similares, este comportamiento puede explicarse por los problemas de escalabilidad relacionados con los acceso a memoria, propios de enfoques de memoria compartida.

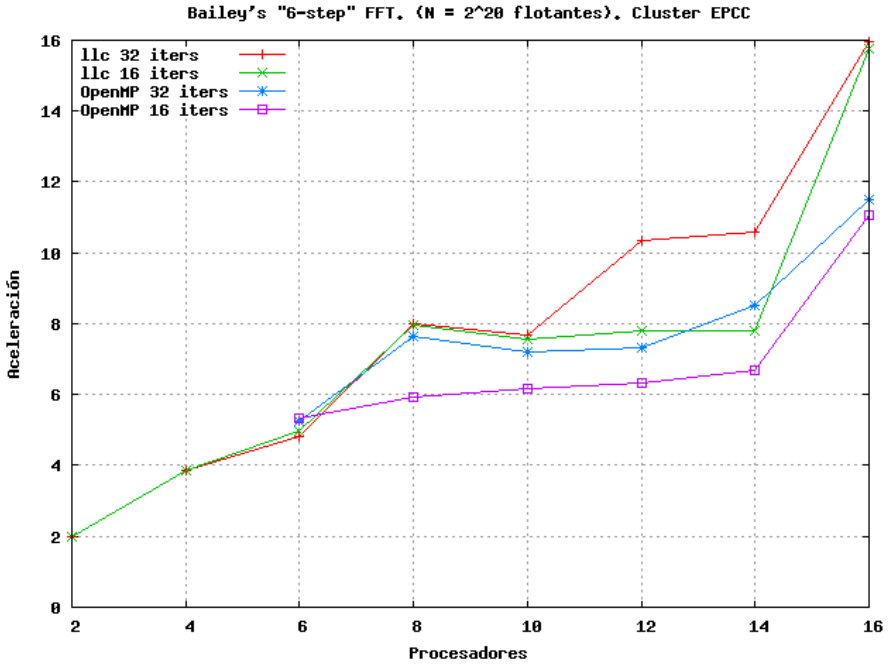

Figura 4.8: Aceleración para el código `FFT` de "6 pasos" de Bailey para `llc` y `OpenMP` en el `Cluster EPCC`

4.2.5. Algoritmo del `cálculo del número` π

El algoritmo de cálculo del número π es frecuentemente uno de los primeros ejemplos que se estudian en cualquier API paralela. El cálculo del valor de π se basa en que éste coincide con el área encerrada bajo la curva $\frac{1}{1+x^2}$ para el intervalo $[0, 1]$, por lo que puede ser calculado según la integral $\int_0^1 \frac{1}{1+x^2} dx = 4\ arctan(x)|_0^1 = \pi$.

El algoritmo divide este intervalo $[0, 1]$ en N segmentos de tamaño w y acumula el área de los rectángulos delimitados por el punto central de cada segmento (t) y la curva del valor en ese punto, tal y como se muestra en la figura 4.9. De este modo, la aproximación del valor de π se obtiene al evaluar

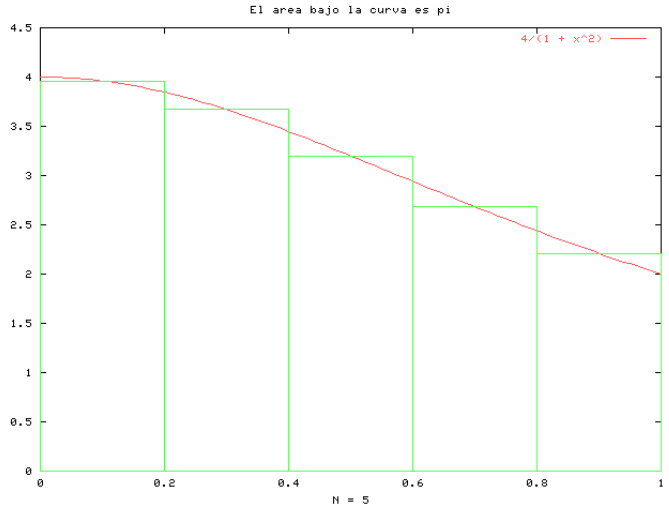

Figura 4.9: Estimación de π

el sumatorio $\pi \approx \sum_{i=0}^{N-1} \frac{4}{N(1+(\frac{i+\frac{1}{2}}{N})^2)}$, siendo el número de iteraciones N el único parámetro que recibe el algoritmo y que establece la precisión de la aproximación.

El código `llc` de este algoritmo ya ha sido presentado en el listado 1.20 (página 49), que muestra la paralelización del algoritmo tomando como partida un código `OpenMP`. En el listado 4.6 mostramos la paralelización equivalente utilizando únicamente directivas específicas de `llc`, lo que permite al lector comparar ambos modos de programar en `llc`.

```
1   w = 1.0 / N;
2   pi = 0.0;

4 #pragma llc parallel for
5 #pragma llc reduce (&pi, &pi_g, 1, LLC_SUM)
6   for (i = 0; i < N; i = i++) {
7     local = (i + 0.5)*w;
8     pi = pi + 4.0/(1.0 + local*local);
9   }
10  pi = pi_g * w;
```

Listado 4.6: Ejemplo del cálculo de π mediante un código `llc`

En la figura 4.10 se presentan los resultados computacionales obtenidos para el algoritmo de cálculo de π en el `Cray T3E` con 10^9 iteraciones. Las

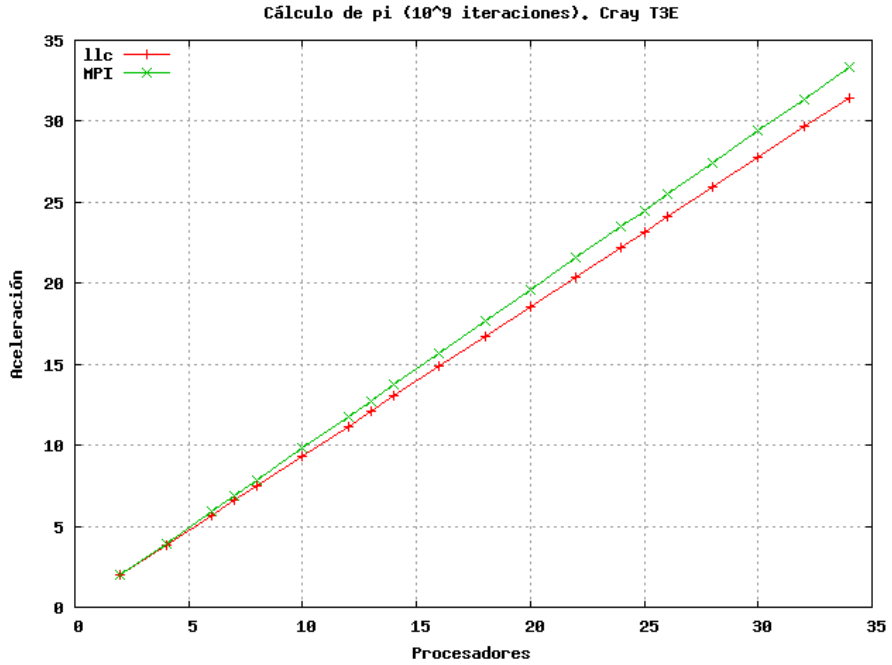

Figura 4.10: Aceleración para el código de cálculo de π en el `Cray T3E`

características del algoritmo avalan la linealidad de los resultados, al ser mínimo el número de comunicaciones que se requieren, transmitiéndose únicamente el resultado tras aplicarse una operación de reducción tipo suma. La aceleración que se obtiene con `MPI` es ligeramente superior a la de `llc`, pero esta leve pérdida de rendimiento queda justificada de sobra con un menor tiempo de desarrollo y depuración. Los tiempos utilizados en el cálculo de las aceleraciones en esta figura se recogen en la tabla 4.4.

La figura 4.11 muestra los resultados de este código en diferentes máquinas, tanto de memoria compartida como distribuida, todos ellos para 10^9 iteraciones. Como se puede observar, los resultados son comparables independientemente del tipo de arquitectura, salvo para el caso de la `SGI O2000`, donde los resultados irregulares obtenidos podrían deberse a una ejecución no exclusiva.

	Cray T3E	
procs.	`llc`	`MPI`
SEQ.	110.33	
2	55.4237	56.2839
4	28.664	28.1396
6	19.4838	18.7603
7	16.7266	16.0808
8	14.665	14.0707
10	11.8168	11.2567
12	9.8714	9.38108
13	9.09699	8.65945
14	8.44845	8.04051
16	7.39958	7.0356
18	6.59856	6.25396
20	5.95015	5.62862
22	5.41453	5.11731
24	4.96429	4.6909
25	4.76043	4.50358
26	4.57615	4.33015
28	4.2524	4.02083
30	3.96855	3.75291
32	3.72043	3.51787
34	3.50705	3.31118

Tabla 4.4: Tiempos (en segundos) obtenidos para el código de cálculo π en el `Cray T3E`

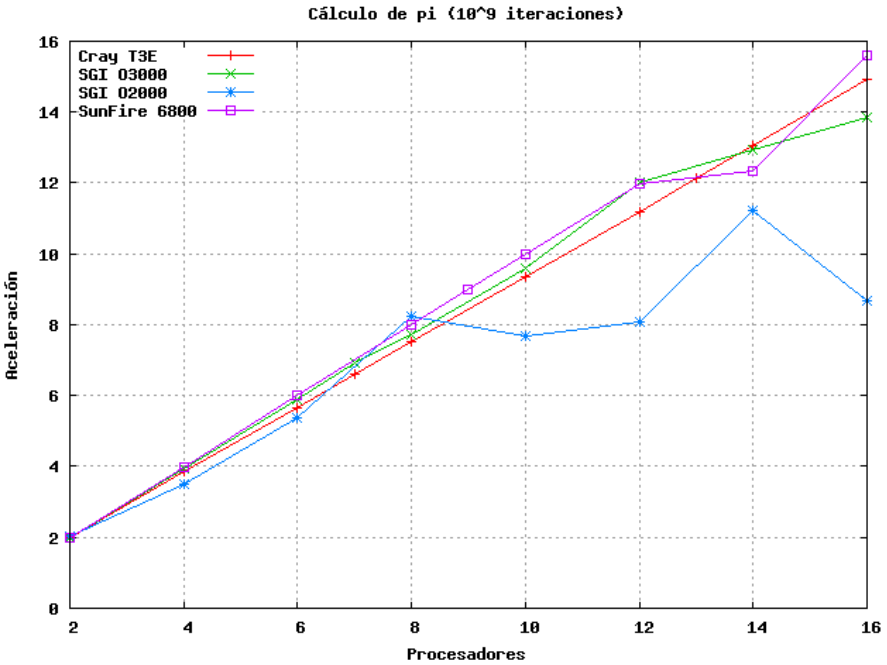

Figura 4.11: Aceleración para el código de cálculo de π en diferentes plataformas

4.2.6. Algoritmo `Embarrassingly Parallel`

El algoritmo `Embarrassingly Parallel` (`EP`) [93] es otro de los algoritmos que hemos extraído de los *NAS Parallel Benchmarks* [112, 9, 89]. Este algoritmo implementa un generador de números aleatorios y su utilidad es típica en muchas de las aplicaciones de simulaciones Monte Carlo. El nombre del algoritmo se debe a que no se requieren comunicaciones para la generación de números aleatorios en sí, por lo que el resultado de la paralelización es un código "embarazosamente" paralelo.

El algoritmo `EP` guarda similitudes con el algoritmo del cálculo de π ya presentado. La mayoría de cálculos necesarios para obtener los números aleatorios se realizan en un bucle principal, siendo el único requisito de comunicación la combinación de los sumatorios al finalizar el bucle. El listado 4.7 presenta una porción de este bucle, pero dado el tamaño total del mismo, en este ejemplo sólo mostramos la porción del código en la que se realizan los cálculos de las desviaciones Gaussianas.

```
#pragma omp for reduction(+:sx,sy) schedule(static)
#pragma llc reduction_type (double, double)
  for (k = 1; k <= np; k++) {
    ...
    for (i = 0; i < NK; i++) {
      x1 = 2.0 * x[2*i] - 1.0;
      x2 = 2.0 * x[2*i+1] - 1.0;
      t1 = pow2(x1) + pow2(x2);
      if (t1 <= 1.0) {
        t2 = sqrt(-2.0 * log(t1) / t1);
        t3 = (x1 * t2);        /* Xi */
        t4 = (x2 * t2);        /* Yi */
        l = max(fabs(t3), fabs(t4));
        qq[l] += 1.0;          /* counts */
        sx = sx + t3;          /* sum of Xi */
        sy = sy + t4;          /* sum of Yi */
      }
    }
    ...
  }
```

Listado 4.7: Bucle principal del algoritmo EP

En este listado 4.7 cabe destacar que las variables que calculan el

sumatorio de X y de Y (`sx` y `sy`, respectivamente), son las únicas variables a comunicar a la finalización del bucle, y previamente se les aplica una operación de reducción tipo suma, tal y como se indica en la directiva de la línea 1. `llc` interpreta y traduce la cláusula de reducción de `OpenMP`, siendo necesario hacer uso de la directiva de `llc reduction_type` de la línea 2 para completar la cláusula de `OpenMP`.

El algoritmo EP brinda un escenario perfecto para paralelización de entornos distribuidos, como `llc`, ya que ofrece un grano considerablemente grueso con un número mínimo de comunicaciones. Por esta razón, podemos tomar este algoritmo como representativo de un caso óptimo de paralelización en el cual se espera alcanzar un rendimiento máximo, lo que permite tener una medida de la eficiencia de nuestra traducción frente a otras alternativas. El algoritmo permite que se indique como parámetro tres tamaños de problema, `A`, `B` y `C`, habiéndose usado este último, que corresponde con el mayor de los tres (2^{32} puntos).

La figura 4.12 presenta la aceleración obtenida para el algoritmo EP en el `Cray T3E`. Los resultados obtenidos confirman las apreciaciones realizadas, alcanzándose una aceleración lineal para todo el rango de procesadores del experimento. La figura 4.12 también recoge los resultados obtenidos mediante un algoritmo *ad hoc* en `MPI`, siendo éstos prácticamente iguales a los de `llc`, lo cual confirma la eficiencia de la traducción realizada por `llc`: para este caso, `llc` presenta unos resultados similares a los de `MPI`, habiéndose invertido un tiempo y esfuerzo considerablemente inferior a la hora de desarrollar el código.

Los resultados de la figura 4.12 quedan ratificados en la figura 4.13, donde hemos ejecutado el mismo experimento sobre el `SunFire 6800`. Otra vez podemos observar la gran similitud de las curvas obtenidas en la mayoría del rango, si bien se observa que `llc` obtiene una pequeña superlinealidad para un número elevado de procesadores, lo que podría ser explicado por una caché ya caliente.

Por último, en la figura 4.14, obtenida en la `SGI O2000`, hemos repetido el experimento añadiendo una ejecución con `OpenMP`. Otra vez se puede constatar la proximidad de los resultados, superponiéndose las curvas en la mayoría de su recorrido.

Los tiempos en los que se basan las figuras 4.13 y 4.14 se recogen en la tabla 4.5, para el `Cray T3E` y la `SunFire 6800` con un tamaño de problema de clase `C` (N = 2^{32}).

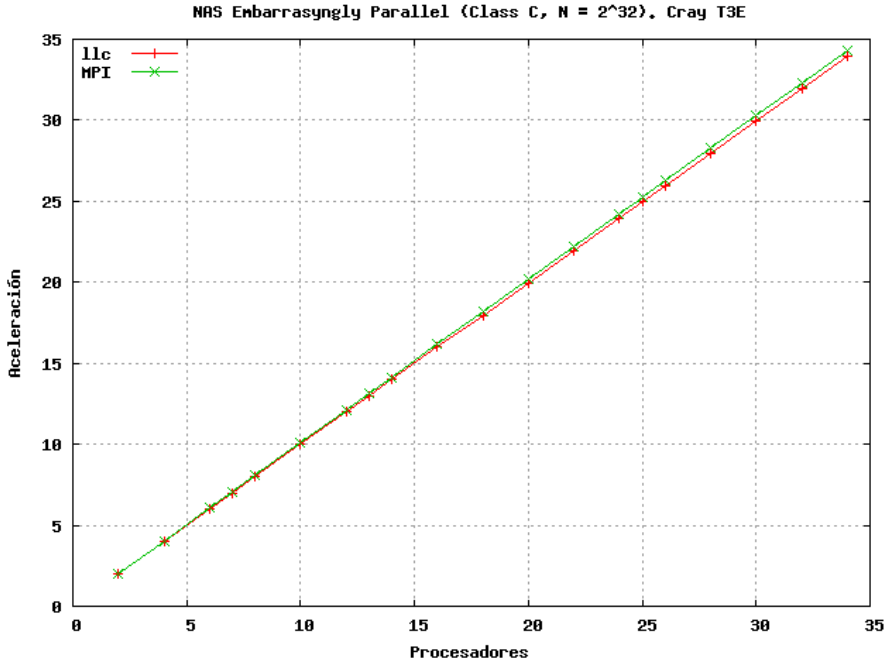

Figura 4.12: Aceleración para el código `EP` en el `Cray T3E`

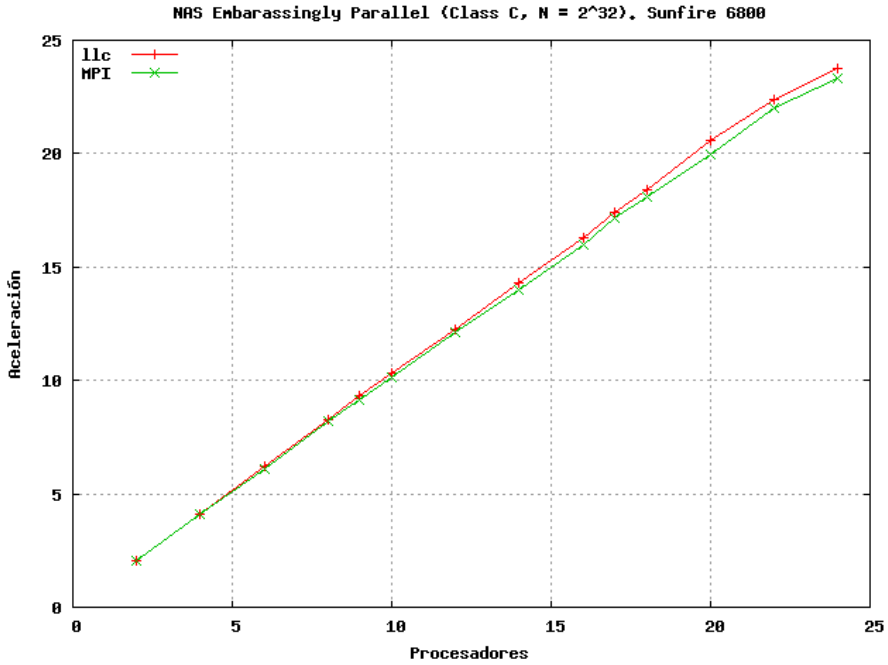

Figura 4.13: Aceleración para el código `EP` en el `SunFire 6800`

	Cray T3E		SunFire 6800		SGI O2000		
procs.	`llc`	`MPI`	`llc`	`MPI`	`llc`	`MPI`	`OpenMP`
SEQ	4534.5		2400		3400		
2	2270.21	2244.63	1160.4	1183.97	1712.05	1698.37	1686.17
4	1135.11	1122.3	581.663	587.635	866.068	859.254	843.143
6	756.774	748.241	387.008	394.805	584.413	586.615	578.065
7	648.726	641.365	289.952				
8	567.592	561.202		292.522	453.882	433.141	426.386
9			257.912	263.105			
10	454.109	448.96	232.125	236.751	344.226	341.923	340.169
12	378.438	374.155	196.259	197.472	313.29	310.284	306.481
13	349.333	345.394					
14	324.39	320.723	167.883	171.199	265.493	263.538	261.793
16	283.802	280.605	147.064	149.996	232.842	225.161	222.222
17			137.983	139.589			
18	252.278	249.428	130.378	132.64	193.242	192.147	190.651
20	227.057	224.496	116.624	120.214	175.589	174.285	173.256
22	206.409	204.077	107.1	109.054	159.512	158.762	156.593
24	189.226	187.092	100.985	102.926	147.529	145.346	144.223
25	181.675	179.623					
26	174.674	172.701			134.158	133.229	132.53
28	162.205	160.377			123.741	123.007	123.124
30	151.397	149.686			116.925	114.882	115.625
32	141.903	140.302			109.764	108.593	108.007
34	133.592	132.077					

Tabla 4.5: Tiempos (en segundos) obtenidos para el código `Embarrassingly Parallel`

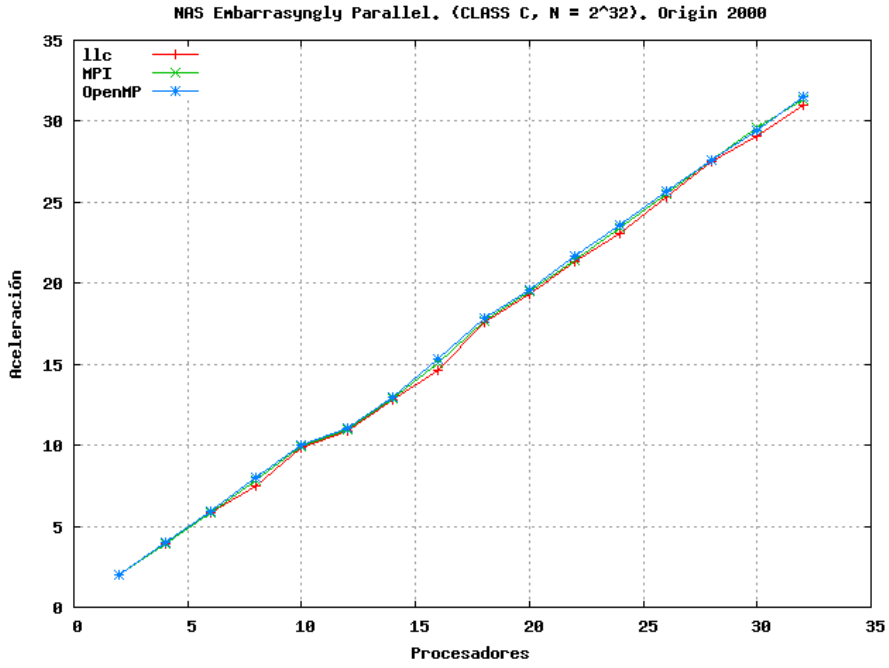

Figura 4.14: Aceleración para el código `EP` en la `SGI O2000`

4.2.7. Algoritmo `Mandelbrot`

El conjunto de Mandelbrot [107] es el dominio de convergencia de la serie compleja definida por la ley de recursión $Z_n = {Z_{n-1}}^2 + C$. El cálculo de su área [108] es una cuestión abierta en matemáticas, utilizándose métodos estadísticos para su cómputo, ya que obtener una estimación precisa por métodos analíticos no es simple. Una forma de estimar esta área es mediante un algoritmo que separe los puntos del plano complejo que pertenecen al conjunto de Mandelbrot (puntos internos) de los que no pertenecen (externos). El algoritmo `Mandelbrot` que hemos utilizado calcula una estimación del área del conjunto mediante una técnica Monte Carlo, que toma como parámetro de entrada el número de puntos (`npoints`) del plano a explorar.

El listado 4.8 muestra el código del bucle principal del algoritmo paralelizado con `llc`. El objetivo de este bucle es determinar cuántos puntos están fuera del conjunto, para luego calcular el número de puntos internos, el área total del mismo y el error obtenido (líneas 22 y 23). Cada iteración corresponde con la comprobación de uno de los puntos del plano, cumpliéndose la condición de la línea 14 para los puntos externos al conjunto,

razón por la que se incrementa la variable `numoutside` que almacena la cantidad de estos puntos (línea 15). Como cada grupo habrá calculado un número parcial de puntos, es necesario aplicar una operación de reducción de tipo suma al comunicar el resultado obtenido. Como en otras situaciones, queda patente la simplicidad del desarrollo, siendo sólo necesario, en este caso, introducir la directiva `llc reduction_type` de la línea 6 para complementar la cláusula de `OpenMP reduction` de la línea 3.

```
1  numoutside = 0;
2  #pragma omp parallel for default(none)      \
3              reduction(+:numoutside)         \
4              private(i,j,ztemp,myid,nlocal, \
5              ilower,z,iupper),shared(nt,c)
6  #pragma llc reduction_type (int)
7  for(i = 0; i < npoints; i++) {
8    z.creal = c[i].creal;
9    z.cimag = c[i].cimag;
10   for (j = 0; j < MAXITER; j++) {
11     ztemp = (z.creal*z.creal)-(z.cimag*z.cimag)+c[i].creal;
12     z.cimag = z.creal * z.cimag * 2 + c[i].cimag;
13     z.creal = ztemp;
14     if (z.creal*z.creal+z.cimag*z.cimag > THRESOLD) {
15       numoutside++;
16       break;
17     }
18   } /* for j */
19 } /* for i */
20 numinside = npoints - numoutside;
21 /* Calculate area and error */
22 area = 2.0 * 2.5 * 1.125 * numinside / npoints;
23 error = area / sqrt(npoints);
```

Listado 4.8: Algoritmo `Mandelbrot` paralelizado con `llc`

El algoritmo `Mandelbrot` presenta unas características similares a las del algoritmo `EP` y del cálculo de π estudiados en apartados previos. No obstante, a pesar de que las comunicaciones también son mínimas y sólo se transmite el resultado de la reducción, en este caso el grano del problema no es tan grueso como el del `EP`, por lo que se espera un rendimiento inferior. Por otro lado, el bucle de la línea 7 puede ser interrumpido por la sentencia `break` (línea 16) si se cumple la condición de la línea 14. Esto produce un desequilibrio de la carga de las tareas que afecta negativamente al rendimiento global de la

aplicación.

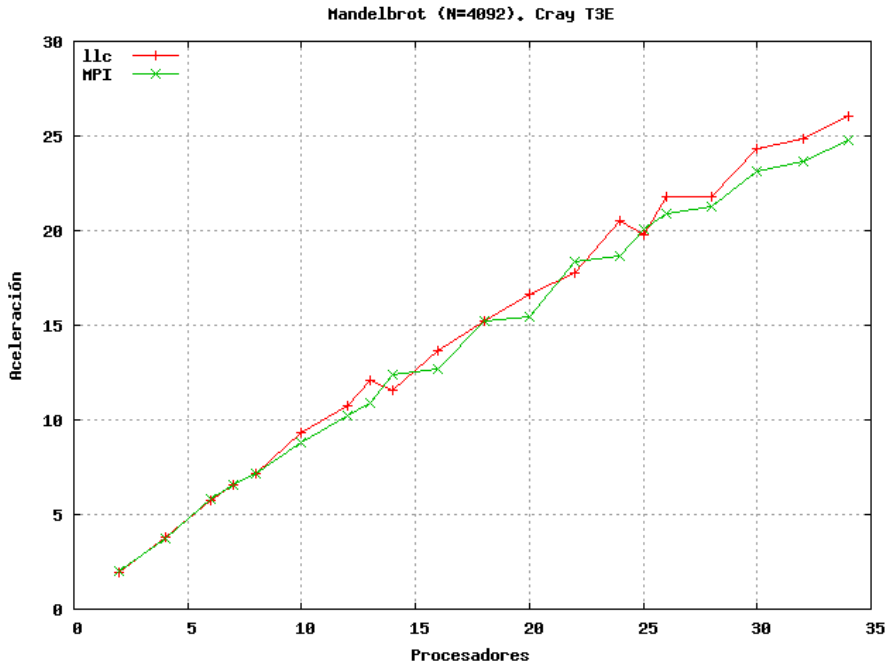

Figura 4.15: Aceleración para el código `Mandelbrot` en el `Cray T3E`

La figura 4.15 obtenida en el `Cray T3E` muestra la aceleración de la versión `llc` comparada con la del algoritmo *ad hoc* en `MPI` para 4092 puntos. Podemos observar que, aunque el comportamiento de las gráficas tiene una tendencia lineal, no se llega a la linealidad del `EP`. Las diferencias entre las versiones `llc` y `MPI` no son destacables y ambas curvas se entrecruzan constantemente.

La figura 4.16 muestra las curvas para la versión `llc` y `MPI` para 4092 puntos en la `SunFire 6800`, añadiendo también una versión `OpenMP`. Del mismo modo que en la anterior figura 4.15, las curvas se entrelazan entre ellas, siendo el rendimiento prácticamente el mismo con las tres aproximaciones. Las aceleraciones de estas dos figuras 4.15 y 4.16 se han calculado a partir de los tiempos que se muestran en la tabla 4.6.

	Cray T3E		SunFire 6800		
procs.	`llc`	`MPI`	`llc`	`MPI`	`OpenMP`
SEQ	7.113		6.28		
2	3.6137	3.58746	3.19216	3.16854	3.19261
4	1.87574	1.89724	1.66339	1.67399	1.65961
6	1.24133	1.21598	1.09655	1.07399	1.10028
7	1.0878	1.08245			
8	0.994022	0.995775	0.976133	0.87961	0.888177
9			0.756387	0.767447	0.759237
10	0.76086	0.808647	0.684115	0.714259	0.676345
12	0.660097	0.695	0.599327	0.613951	0.588772
13	0.587023	0.654953			
14	0.613153	0.574802	0.543063	0.508063	0.547681
16	0.520333	0.561457	0.450778	0.49621	0.465232
17			0.435266	0.449693	0.430334
18	0.466721	0.467757	0.41417	0.41418	0.419202
20	0.426783	0.46106	0.377531	0.408019	0.383949
22	0.400615	0.387776			
24	0.347104	0.381079			
25	0.359635	0.354164			
26	0.326689	0.340901			
28	0.326857	0.334123			
30	0.292493	0.307744			
32	0.286637	0.300952			
34	0.273005	0.287461			

Tabla 4.6: Tiempos (en segundos) obtenidos para el código `Dinámica Molecular`

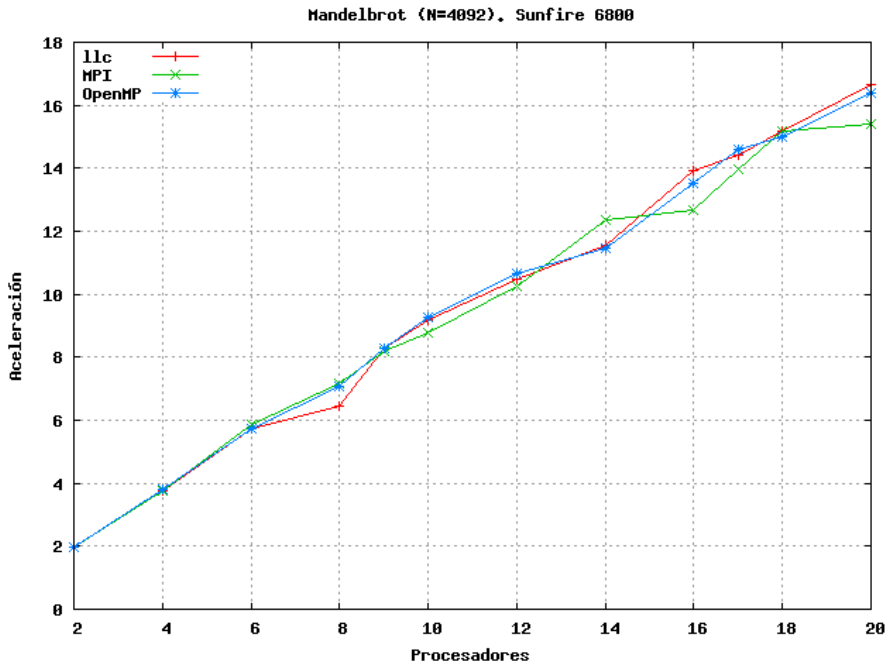

Figura 4.16: Aceleración para el código `Mandelbrot` en el `SunFire 6800`

4.2.8. Algoritmo `Microlensing`

Los campos gravitatorios masivos, como los de las galaxias, inducen efectos ópticos tales como la formación de múltiples imágenes (gravitational lensing). Dada la naturaleza granular de las galaxias (la materia en las galaxias no está distribuida uniformemente), el movimiento de las estrellas que se interponen en el camino de la luz puede producir un pequeño, pero medible, efecto de lente gravitacional (*microlensing*) [13]. Este efecto se traduce en pequeñas variaciones en el brillo del objeto observado.

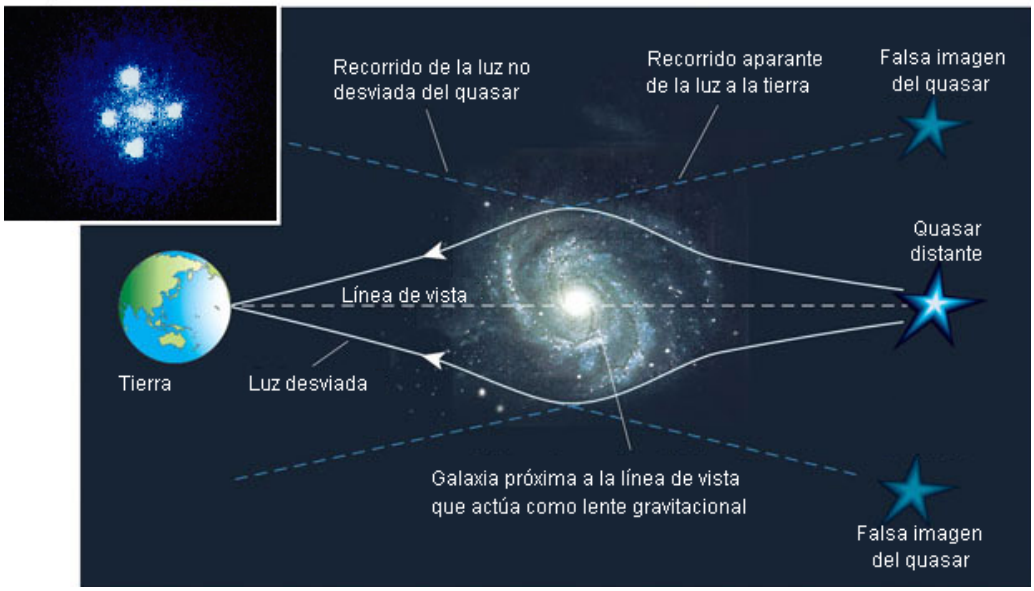

Figura 4.17: Esquema del efecto *Microlensing* y cúadruple quasar Q2237+0305

La figura 4.17 muestra un esquema de este efecto. En la esquina superior izquierda de esta figura se puede observar una imagen del cúadruple quasar Q2237+0305, conocido como la *Cruz de Einstein*, uno de los mejores ejemplos de este fenómeno que se conoce hasta la fecha.

Los patrones de magnificación representan los cambios de magnitud asociados a diferentes configuraciones de estrellas en la galaxia lente y de desplazamientos relativos de dicha fuente. La comparación de los patrones generados con la variabilidad observada permite obtener información tanto del objeto lejano como de la galaxia lente. [75].

Presentamos este algoritmo como un caso de estudio del rendimiento obtenible según sea el nivel del bucle que se paraleliza en el caso de contar

con uno o más bucles anidados. El listado 4.9 muestra el código principal que realizan los cálculos del algoritmo de `microlensing`. Como se puede observar, existen dos bucles anidados, el externo de la línea 3 y el interno de la línea 13, cuyo código puede consultarse en el listado 4.10 (existe un tercer bucle, en la línea 6, pero no lo tendremos en cuenta en esta paralelización debido a un grano inferior comparado con el resto de bucles).

En este primer listado 4.9 se ha optado por realizar la paralelización del bucle externo (línea 3). Para ello es necesario comunicar la variable `s` que guarda el resultado del patrón de amplificación y que requiere una operación de reducción. Al ser `s` un vector, no es posible usar la cláusula de reducciones de `OpenMP`, ya que sólo permite variables escalares. En su lugar, en esta ocasión hemos indicado la operación de reducción mediante la directiva específica de `llc reduce` (línea 2), ya que permite realizar operaciones de reducción elemento a elemento sobre vectores a partir de la posición dada y para el número de elementos indicado, que este caso son `size` elementos.

```
1  #pragma llc for
2  #pragma llc reduce (s, s_aux, size, LLC_SUM)
3    for (i = 0; i < nrayx_i; ++i) {
4      xx = -lplsx + (float)i * dist;

6      for ( isum = 0; isum < nlen; ++isum ) {
7        x = xx - xpos[isum];
8        field[isum].sqrx = x * x;
9        field[isum].xrat = x * rat[isum];
10     }
11     kgx = kg1 * xx - xrfmin;

13     for (j = 0; j < nrayy_i; ++j) {
14       ...
15     }
16   }
```

Listado 4.9: Paralelización de bucle externo

El listado 4.10 presenta la paralelización del bucle interno (el bucle de la línea 3 de este listado se corresponde con el que aparece en la línea 13 del listado 4.9). Una característica de este bucle es que la carga computacional de cada iteración no es constante, sino que depende de la evaluación de la condición de las líneas 14 y 15, lo que produce una distribución de cargas asimétricas. Por otro lado, existe un tercer bucle anidado, en la línea 7,

aunque para este estudio sólo tendremos en cuenta dos niveles de anidamiento en cuanto se refiere a su paralelización.

```
#pragma llc for
#pragma llc result (&pos[j], 1)
  for (j = 0; j < nrayy_i; ++j) {
    ysum = pos2[j].xyk;
    xsum = kgx;
    temp = pos2[j].xy;
    for (isum = 0; isum < nlen; ++isum) {
      y = temp - ypos[isum];
      temp_modu = field[isum].sqrx + y * y;
      ysum -= (y * rat[isum]) / temp_modu;
      xsum -= field[isum].xrat / temp_modu;
    }

    if (ysum >= zeroYround && ysum < sizeYround &&
            xsum >= zeroXround && xsum < sizeXround) {
      ipix2 =  ysum  / ybin;
      ipix1 =  xsum  / xbin;
      x1 =(int)(ipix1+offSetX); y1 =(int)(ipix2+offSetX);
      x2 =(int)(ipix1-offSetY); y2 =(int)(ipix2-offSetY);
      hx1 = fabs(ipix1 - x1); hx2 = 1.0 - hx1;
      hy1 = fabs(ipix2 - y1); hy2 = 1.0 - hy1;
      y1size = y1 * sizeX;
      y2size = y2 * sizeX;
      pos[j].pos0 = x1 + y1size; pos[j].val0 = hx1 * hy1;
      pos[j].pos1 = x1 + y2size; pos[j].val1 = hx1 * hy2;
      pos[j].pos2 = x2 + y1size; pos[j].val2 = hx2 * hy1;
      pos[j].pos3 = x2 + y2size; pos[j].val3 = hx2 * hy2;
    }
  }
```

Listado 4.10: Paralelización de bucle interno

Cuando existen varios bucles anidados susceptibles de ser paralelizados, también existe la posibilidad de utilizar *paralelismo multinivel*. Sin embargo, el paralelismo de varios niveles no siempre está disponible, como en el caso de `OpenMP`, que a pesar de que la especificación sí permita el paralelismo multinivel, en la actualidad, la mayoría de compiladores de este lenguaje simplemente serializan cualquier región paralela anidada [79, 139]. `llc`, al contrario, sí ofrece soporte a paralelismo multinivel [55], mediante el

anidamiento de la mayoría de sus directivas paralelas, recursividad, etc.

En `llc`, el paralelismo multinivel se justifica a través del modelo `OTOSP` y su concepto de conjuntos de procesadores. Un conjunto de procesadores de un nivel puede subdividirse en varios subconjuntos para dar lugar a paralelismo en el siguiente nivel, y así sucesivamente. No obstante, el paralelismo multinivel sólo tiene sentido siempre y cuando el conjunto tenga más de un procesador, ya que un único procesador no puede soportar niveles de paralelismo adicionales, y la ejecución tendrá que ser serializada a partir de ese nivel. Esto implica que, para poder aplicar eficientemente el paralelismo multinivel en `llc`, el número de procesadores del conjunto deba ser suficientemente superior al número de tareas a ejecutar.

En el listado 4.10 el bucle del primer nivel (línea 3) ejecuta `nrayx_i` iteraciones, correspondiendo cada iteración con una tarea. El valor de `nrayx_i` ha sido del orden de medio millar para los experimentos computacionales realizados. Esto implica que sería necesario ejecutar estos experimentos en una máquina paralela con varios miles de procesadores para poder explotar el paralelismo multinivel. Como una máquina de estas características no está disponible en el conjunto de equipos a los que hemos tenido acceso, se ha desestimado la posibilidad de aplicar paralelismo anidado en este algoritmo, concepto que será considerado en próximos apartados.

La figura 4.18 muestra los resultados computacionales obtenidos [103] al procesar una matriz de tamaño 512×512 pixels donde se lanzan 16×16 rayos por pixel, siendo el trazado de cada rayo influenciado por (aproximadamente) 3500 estrellas, lo que implica la resolución de una ecuación por estrella, del orden de 0.23 billones de ecuaciones totales.

Esta figura 4.18 compara los resultados obtenidos en la `IBM RS-6000` al paralelizar el bucle interno y externo, tanto para `llc` como para un código *ad hoc* en `MPI`. En estos resultados observamos cómo el mejor rendimiento se obtiene para la versión que paraleliza el bucle externo, tanto para `llc` como para `MPI`, superponiéndose ambas curvas para todo el rango de procesadores del experimento. Esto se debe a que esta versión es la que obtiene un mayor grano de aplicación, a la vez que minimiza las comunicaciones. Estas circunstancias son las que favorecen el rendimiento de aplicaciones de memoria distribuidas del tipo de `llc` o `MPI`.

En cuanto a los resultados obtenidos al paralelizar el bucle interno, podemos observar como tienen un peor comportamiento que el bucle externo, tendiendo a distanciarse ambas gráficas a medida que el número de procesadores aumenta. Este comportamiento es más destacable en el caso de `llc`, debido a que se dan las condiciones contrarias que en el caso anterior, un

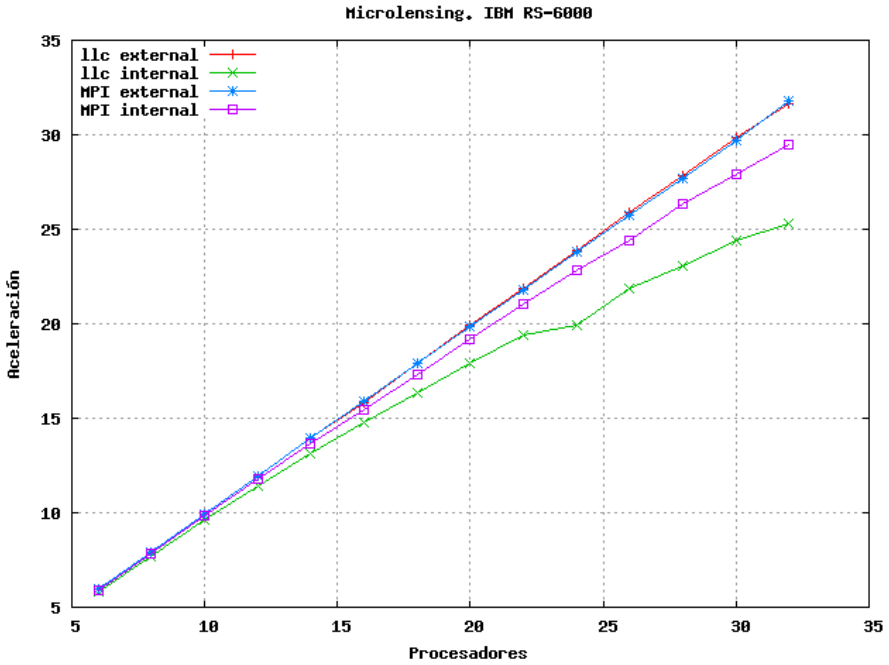

Figura 4.18: Aceleración para el código `microlensing` en el `IBM RS-6000`

grano más fino y un mayor número de comunicaciones (si bien de un tamaño menor), lo cual no favorece al caso de memoria distribuida.

La figura 4.19 [103] muestra el mismo experimento, realizado esta vez en el `Bull NovaScale Server`, recogiéndose los tiempos obtenidos en la tabla 4.7. En esta ocasión hemos incluido una versión `OpenMP` de la paralelización del bucle interno, aunque sólo hasta 16 procesadores por limitaciones en las particiones de procesadores para ejecución en memoria compartida. Los resultados son similares a los estudiados anteriormente, si bien la paralelización del bucle interno de `llc` presenta un rendimiento inferior en esta situación con respecto a `MPI`. Una razón que lo justifica es que la versión `llc` se ve más afectada cuando la distribución de carga de cada tarea presenta un marcado desequilibrio, puesto que existe un punto de sincronización implícito al finalizar la ejecución del bucle paralelo, momento en el que los resultados se comunican.

	Bull NovaScale				
procs.	`llc ext`	`MPI ext`	`llc int`	`MPI int`	`omp int`
SEQ	17731.0				
4	4442.76	4728.58	4596.10	4607.15	4782.20
6	3155.55	3156.60	3302.36	3075.68	3181.09
8	2362.17	2367.65	2542.60	2315.91	2394.31
10	1889.95	1896.12	2100.32	1858.77	1922.08
12	1576.68	1577.97	1765.91	1553.26	1609.78
14	1350.92	1350.00	1578.51	1338.92	1384.57
16	1184.12	1183.79	1425.23	1175.96	1220.73
18	1051.21	1051.68	1286.82	1050.84	
20	946.69	946.24	1178.97	948.21	
22	860.75	861.87	1106.55	864.94	
24	789.00	789.36	1047.48	796.50	
26	728.52	728.97	989.98	736.27	
28	676.52	677.17	965.61	687.35	
30	631.53	632.45	912.70	644.21	
32	592.13	592.10	872.20	610.78	

Tabla 4.7: Tiempos (en segundos) obtenidos para el código `microlensing` en el `Bull NovaScale Server`

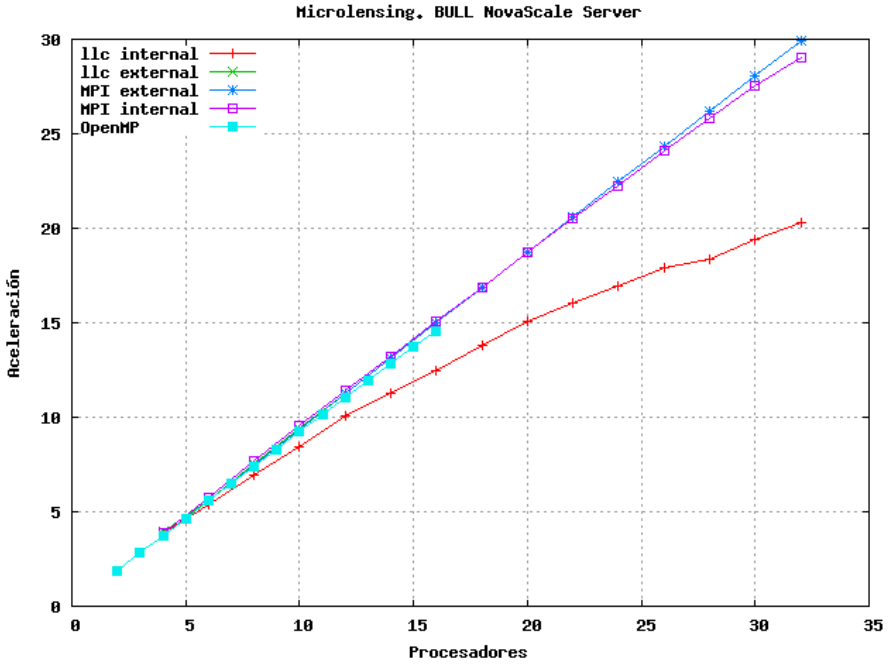

Figura 4.19: Aceleración para el código `microlensing` en el `Bull NovaScale Server`

4.2.9. Algoritmo de `multiplicación de matrices`

El código de multiplicación de matrices (`matrix`) que se presenta en el listado 4.11 se basa en el método tradicional compuesto por tres bucles (líneas 6 a 9). A éstos se les ha añadido un cuarto bucle, en la línea 1, que genera *tareas*, consistiendo cada tarea en la multiplicación de un par de matrices.

El número de filas de la matriz A (columnas en B) es fijo (m), mientras que el número de columnas de A (filas de B) varía en cada tarea t, viniendo determinado el número de columnas de la tarea t-ésima por la expresión $n_t = n + s \times t$, produciendo en todos los casos matrices resultado de tamaño $m \times m$. El cálculo del número de columnas para la tarea actual se realiza entre las líneas 2 y 5, calculando además el valor de los punteros dependiendo de la tarea, y es común para todos los listados de este apartado.

```
1   for(t = 0; t <= tasks - 1; t++) {
2     col = n + s * t;
3     A = AA + m * ((n * t) + (s * ((t * (t - 1)) / 2)));
4     B = BB + m * ((n * t) + (s * ((t * (t - 1)) / 2)));
5     C = CC + m * m * t;
6     for(i = 0; i <= m - 1; i++) {
7       for(j = 0; j <= m - 1; j++)
8         for(*(C+i*m+j) = 0.0, k = 0; k <= col - 1; k++)
9           *(C+i*m+j) += (*(A+i*col+k) * *(B+k*m+j));
10    }
11  }
```

Listado 4.11: Bucles de multiplicación de matrices a paralelizar

Este código se propone como un escenario para contrastar el rendimiento de las paralelizaciones obtenidas con `llc` dependiendo de la profundidad del bucle que es paralelizado y de la utilización de paralelismo multinivel.

El listado 4.12 muestra la paralelización de un único nivel en el bucle externo, encargado de generar las tareas. Esta paralelización ofrece el grano más grueso posible y un número mínimo de comunicaciones (una por tarea), pero de mayor tamaño (se comunica la matriz resultado completa, tamaño $m \times m$, línea 3). Cabe mencionar que estos datos sólo pueden ser interpretadas a nivel teórico, ya que al poseer `llc` estrategias de reducción de comunicaciones mediante empaquetado de datos, el número de comunicaciones y su tamaño puede variar según los parámetros de ejecución y la política de asignación de grupos empleada.

En este listado 4.12 destacamos el uso de la directiva `weight` (línea 2), que se utiliza para una distribución más eficiente de los grupos de procesadores. En este caso, puesto que cada tarea producirá matrices con diferente número de columnas (línea 5) y el peso de la tarea está relacionado directamente con este número de columnas, hemos elegido una relación lineal con este valor como estimación del peso.

A nivel conceptual, el uso de la directiva `weight` sólo tendría sentido en casos de paralelismo multinivel, cuando los grupos que ejecutan las tareas más pesadas disponen de más procesadores para lograr así más niveles de paralelismo. A pesar de que en esta primera paralelización sólo se tenga un nivel de paralelismo, el uso de la directiva `weight` tiene otras ventajas asociadas, como unas comunicaciones más eficientes: `llCoMP` realiza las comunicaciones de resultados en paralelo, como se estudió en el subapartado 3.5.2.3, por lo que es interesante que aquellos grupos con las tareas más grandes y más datos a comunicar, posean el mayor número de procesadores posible, para reducir así el tiempo total necesario para difundir los resultados.

```
1  #pragma omp parallel for
2  #pragma llc weight (n + s * t)
3  #pragma llc result(CC + t * m * m, m * m)
4    for(t = 0; t <= tasks - 1; t++) {
5      col = n + s * t;
6      ...
7      for(i = 0; i <= m - 1; i++) {
8        for(j = 0; j <= m - 1; j++)
9          for(*(C+i*m+j) = 0.0, k = 0; k <= col - 1; k++)
10           *(C+i*m+j) += (*(A+i*col+k) * *(B+k*m+j));
11     }
12   }
```

Listado 4.12: Paralelismo de un nivel en el bucle de las tareas

A continuación, en el listado 4.13 procedemos a trasladar la paralelización de un nivel sobre el bucle que recorre la dimensión i. De esta forma, aumentamos en un nivel la profundidad del bucle paralelizado, reduciendo el grano del problema y disminuyendo el tamaño de las comunicaciones, si bien el número total de éstas aumenta (se deben comunicar los m elementos de la dimensión i, línea 4). En este caso no hemos utilizado directiva de equilibrado de carga, ya que al tener todas las tareas igual peso, no aportaría mejora alguna.

```
1   for(t = 0; t <= tasks - 1; t++) {
2     ...
3 #pragma omp parallel for
4 #pragma llc result(C + i * m, m)
5     for(i = 0; i <= m - 1; i++) {
6       for(j = 0; j <= m - 1; j++)
7         for(*(C+i*m+j) = 0.0, k = 0; k <= col - 1; k++)
8           *(C+i*m+j) += (*(A+i*col+k) * *(B+k*m+j));
9     }
10  }
```

Listado 4.13: Paralelismo de un nivel en el bucle i

Para finalizar el estudio del paralelismo uninivel, aumentamos la profundidad del bucle paralelizado, utilizando esta vez el bucle que recorre la dimensión j. En el listado 4.14 se puede observar el resultado de esta paralelización, similar a la anterior. En este nivel se obtiene el grano más fino y el mayor número de comunicaciones, si bien cada una de ellas sólo transfiere un único elemento (línea 5).

```
1   for(t = 0; t <= tasks - 1; t++) {
2     ...
3     for(i = 0; i <= m - 1; i++) {
4 #pragma omp parallel for
5 #pragma llc result(C + i * m + j, 1)
6       for(j = 0; j <= m - 1; j++) {
7         for(*(C+i*m+j) = 0.0, k = 0; k <= col - 1; k++)
8           *(C+i*m+j) += (*(A+i*col+k) * *(B+k*m+j));
9       }
10    }
11  }
```

Listado 4.14: Paralelismo de un nivel en el bucle j

Por último, en el listado 4.15 realizamos una paralelización de dos niveles, aplicando las correspondientes directivas al bucle de las tareas (línea 4) y al bucle que recorre la dimensión i (línea 8). Las explicaciones de las directivas usadas para la paralelización de cada bucle ya han sido presentadas en las discusiones de los listados 4.12 y 4.13, respectivamente. La directiva `weight` (línea 2) permite una gestión más eficiente del paralelismo multinivel, al asignar más recursos a las tareas más pesadas.

Aparte de su elegancia conceptual, el paralelismo multinivel permite una mejor gestión de los procesadores, al conservar un grano relativamente grande y evitar situaciones en las de que más de un procesador ejecuta la misma tarea si el número de éstas es menor que el de procesadores del grupo actual.

```
#pragma omp parallel for
#pragma llc weight (n + s * t)
#pragma llc result(CC + t * m * m, m * m)
  for(t = 0; t <= tasks - 1; t++) {
    ...
#pragma omp parallel for
#pragma llc result(C + i * m, m)
    for(i = 0; i <= m - 1; i++) {
      for(j = 0; j <= m - 1; j++)
        for(*(C+i*m+j) = 0.0, k = 0; k <= col - 1; k++)
          *(C+i*m+j) += (*(A+i*col+k) * *(B+k*m+j));
    }
  }
```

Listado 4.15: Paralelización de dos niveles

Para que el paralelismo multinivel sea efectivo, tendremos que asegurar que el número de procesadores sea suficientemente mayor que el de las tareas. Por ello, en los resultados computacionales se ha elegido un número relativamente bajo de tareas ($T = 3$). Las matrices tienen dimensiones $n = 800$, $m = 500$ y la variación del número de columnas viene dada por $s = 5$. Además, para reducir fluctuaciones, el algoritmo se repite tres veces, tomándose la media de los tiempos para el cálculo de la aceleración.

El primero de los resultados computacionales para este algoritmo se muestra en la figura 4.20 y los datos fueron tomados en la `SGI O3000`, estando recogidos los tiempos en la tabla 4.8. En esta figura se puede comprobar que la aceleración máxima que se puede obtener en el bucle de tareas es de 3, puesto que $T = 3$. Añadir más procesadores no mejora la aceleración, ya que ésta ha alcanzado su techo, sino que aumenta las comunicaciones y ello puede producir una reducción del rendimiento.

La paralelización del bucle de la dimensión i ofrece unos resultados prácticamente lineales, si bien para un número alto de procesadores se produce una desaceleración debida al incremento de las comunicaciones. Este factor es más evidente en la paralelización de la dimensión j, donde la curva de aceleración presenta un comportamiento logarítmico.

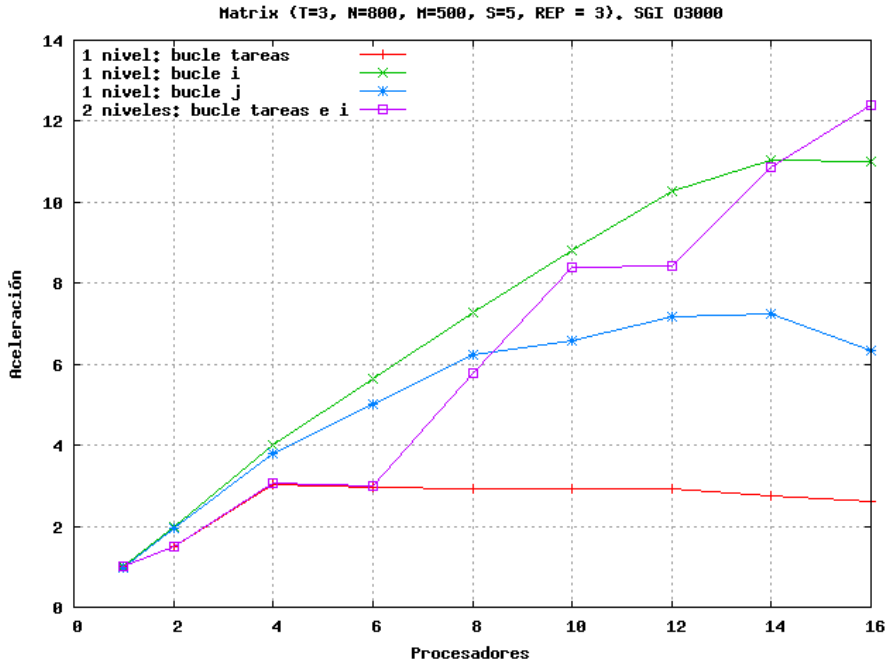

Figura 4.20: Aceleración para el código `matrix` en la `SGI O3000`

Para esta ejecución, el mejor rendimiento se obtiene al usar paralelismo multinivel, paralelizando simultáneamente el bucle de tareas y el de la dimensión i. Destaca que en esta curva existan unas *zonas de aceleración constante*, como las que se pueden apreciar entre 4 y 6 procesadores y entre 10 y 12. Este fenómeno es causado por el hecho de que sólo se consigue aumentar el rendimiento cuando aumenta o bien el número de subgrupos paralelos o bien la profundidad del paralelismo multinivel[1].

Al tener en este caso tres iteraciones el bucle externo, es de esperar que el crecimiento de la aceleración se produzca en la mayoría de casos para múltiplos de tres procesadores, si bien este comportamiento no puede ser predicho de antemano ya que el uso de la directiva de pesos repercute en una distribución no equitativa del número de procesadores en cada grupo. Para una mejor comprensión de este comportamiento, el lector puede consultar la figura 2.1 (página 55), que muestra la distribución jerárquica de los procesadores para el caso de paralelismo multinivel en el modelo OTOSP.

La figura 4.21 presenta los resultados obtenidos con los mismos parámetros que en el caso anterior, en el `Cluster ULL`, utilizando la red

[1]En este algoritmo no es posible aumentar esta profundidad porque sólo existen dos bucles anidados.

`Infiniband` para las comunicaciones. El comportamiento es similar al de la figura 4.20, si bien en este caso el mejor comportamiento se obtiene para la paralelización del bucle j con un número bajo de procesadores, rendimiento que se ve reducido al aumentar el número de los mismos, adquiriendo un comportamiento logarítmico. Los resultados obtenidos para la paralelización del bucle i y el paralelismo de dos niveles son similares, observándose los tramos de aceleración constante característicos del paralelismo multinivel.

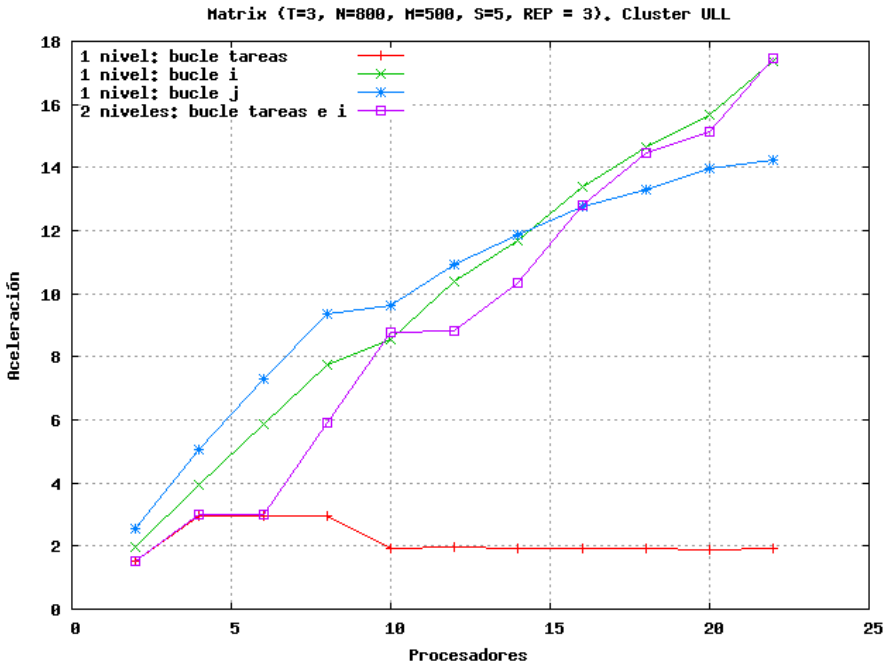

Figura 4.21: Aceleración para el código de multiplicación de matrices en el `Cluster ULL`

Un hecho a destacar del `Cluster ULL` es que es un sistema heterogéneo compuesto por procesadores de diferentes velocidades (consultar subapartado A.2.7 en la página 256). Esta situación provoca algunas alteraciones en los resultados. Por ejemplo, podemos señalar el cambio de comportamiento que se puede observar el las curvas de la figura 4.21 cuando se utilizan más de 8 procesadores, ya que en este momento entran en uso procesadores de diferente velocidad. Este hecho queda patente en la curva de la paralelización del bucle de tareas, donde la aceleración límite de valor tres se ve reducida a un valor dos cuando se utilizan procesadores de diferente velocidad, ya que los más rápidos deben "esperar" a que finalice la ejecución de los procesadores más lentos a la hora de realizar la comunicación de resultados.

	SGI O3000			
procs.	bucle `t`	bucle `i`	bucle `j`	bucles `t-i`
SEQ	12.50			
2	8.28846	6.32016	6.44403	8.28875
4	4.22066	3.22178	3.39142	4.17758
6	4.2084	2.20473	2.47088	4.13483
8	4.27135	1.70376	1.98811	2.14779
10	4.2348	1.4155	1.8938	1.48597
12	4.24668	1.21381	1.73807	1.4787
14	4.5128	1.12541	1.71215	1.14409
16	4.79195	1.1385	1.97168	1.00989

Tabla 4.8: Tiempos (en segundos) obtenidos para el código de multiplicación de matrices

4.2.10. Algoritmo `quickHull`

Una vez estudiado el paralelismo multinivel a través del anidamiento explícito de constructos paralelos de `llc`, como en el listado 4.15 del apartado 4.2.9 dedicado al algoritmo de multiplicación de matrices, en este apartado examinaremos el uso de la *recursividad* para explotar más de un nivel de paralelismo.

Para ello, como primer ejemplo utilizaremos el algoritmo `quickHull` [20] que calcula la *envolvente convexa* de una nube de N puntos, es decir, aquellos puntos que se encuentran en el perímetro de la menor región convexa que los contiene a todos. El funcionamiento general es el siguiente:

1. Se seleccionan los dos puntos con mayor distancia euclídea entre sí (puntos `min_x` y `max_x` de la figura 4.22). Estos dos puntos pertenecerán obligatoriamente a la envolvente. Se traza una arista entre ellos, lo que divide el problema en dos subconjuntos de puntos.

2. Para cada subconjunto, se busca el punto con mayor distancia a la arista trazada en el paso anterior (en la figura 4.22, estos puntos son `min_y` para el subconjunto inferior y `max_y` para el superior).

3. El punto obtenido en el paso anterior pertenece a la envolvente y se trazan dos aristas desde este nuevo punto hasta los dos vértices de la arista original. De esta forma se crea un triángulo por subconjunto (en la figura 4.22 se tiene el triángulo con vértices `min_x`, `max_x` y `max_y` para el subconjunto superior y `min_x`, `max_x` y `min_y` para el inferior). Los

puntos situados en el interior de estos triángulos no podrán pertenecer a la envolvente, por lo que las nuevas búsquedas sólo tendrán en cuenta a los puntos exteriores.

4. Se repiten los dos pasos anteriores, creándose con cada nuevo triángulo dos subconjuntos independientes y paralelizables, uno por cada nueva arista. El algoritmo finaliza cuando no existen más puntos exteriores.

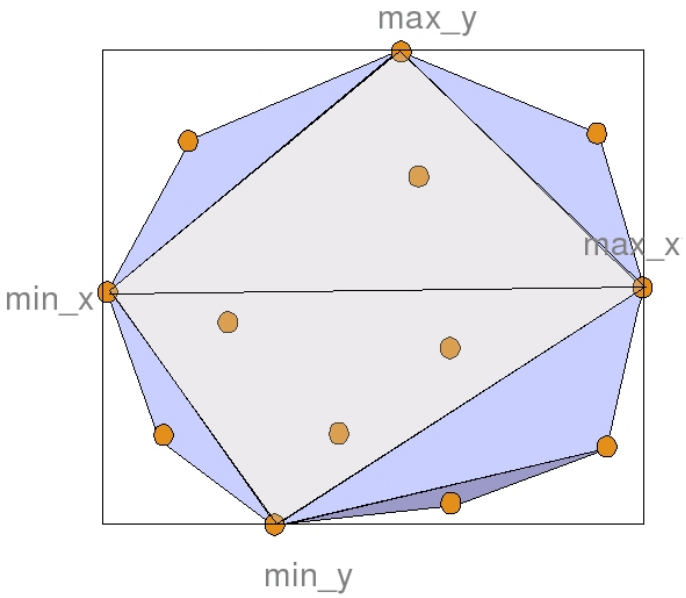

Figura 4.22: Envolvente convexa

Para el caso bidimensional, que es el que nos ocupa, el resultado del algoritmo es un polígono convexo como el que se muestra en la figura 4.22. De esta figura podemos extraer dos observaciones: por un lado, colores más intensos indican etapas más avanzadas y, por otro lado, todos aquellos subconjuntos con el mismo color pueden ser computados en paralelo.

El código del listado 4.16 presenta un bucle paralelizado del algoritmo *divide y vencerás* `quickHull`. En la línea 10 se puede observar como el bucle divide en dos el conjunto de puntos actual para evaluar en paralelo cada uno de los subconjuntos mediante la llamada recursiva a la función `forall_q_hull()` de la línea 11. El tamaño de los subconjuntos es variable, como se desprende de los parámetros utilizados para comunicar los resultados en la directiva de la línea 9.

Las características del algoritmo ocasionan la división en dos partes de los conjuntos de procesadores, siendo cada parte evaluada en paralelo. Esto

```
1  void forall_q_hull(intType **f_hull, intType **l_hull,
2                     intType *m, intType *size) {
3    ...
4    if (max != NULL){
5      partition(firstV[0], lastV[1], max,
6              &lastV[0], &firstV[1], &maxV[0], &maxV[1]);

8  #pragma omp parallel for
9  #pragma llc result (&sizeV[i], 1)
10     for (i = 0; i < 2; i++) {
11      forall_q_hull(&firstV[i],&lastV[i],maxV[i],&sizeV[i]);
12     }
13     ...
14   }
15 }
```

Listado 4.16: Bucle paralelizado en el algoritmo `quickHull`

crea un árbol binario en el que es necesario disponer de un número de procesadores potencia de dos para poder descender un nivel y aumentar de este modo el rendimiento. Este comportamiento puede comprobarse en la la figura 4.23 que muestra la aceleración del algoritmo `quickHull` obtenida en la `SGI O3000` para ejecuciones realizadas con 2, 4 y 8 $\times 2^{20}$ puntos distribuidos en un semicírculo, de modo que se obtiene el caso computacionalmente más complejo donde todos los puntos pertenecen a la envolvente. En esta figura puede observarse que la aceleración aumenta para valores de procesadores potencia de 2 (2, 4, 8 y 16), manteniéndose prácticamente constante para un número de procesadores situado entre estos extremos.

La figura 4.24 muestra el resultado obtenido al repetir el experimento con los mismos parámetros en el `Cluster ULL` usando una red `Infiniband`. En estos resultados también queda patente cómo la aceleración sólo crece para valores potencia de dos, si bien el hecho de que el cluster esté formado por procesadores de diferente velocidad[2] produce unos resultados más variables.

La tabla 4.9 recoge los tiempos empleados para obtener las aceleraciones mostradas en las figuras 4.23 y 4.24

[2] véase la tabla A.3 de la página 260.

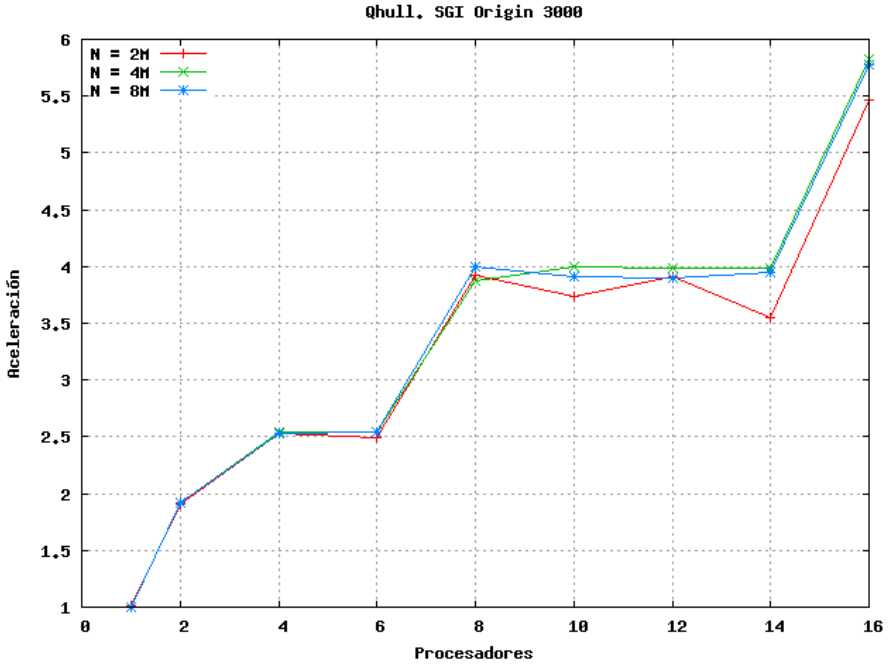

Figura 4.23: Aceleración para el código `quickHull` en la `SGI O3000`

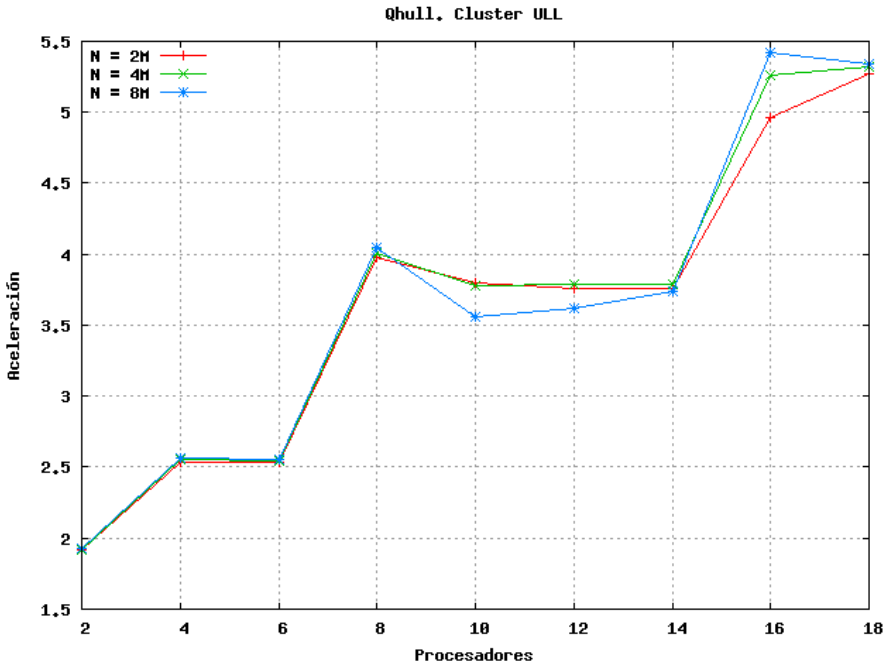

Figura 4.24: Aceleración para el código `quickHull` en el `Cluster ULL`

	SGI O3000			Cluster ULL		
procs.	N = 2M	N = 4M	N = 8M	N = 2M	N = 4M	N = 8M
SEQ	64.5566	134.663	281.979	9.78451	20.3980	42.4607
2	33.6785	70.269	146.913	5.09332	10.6305	22.0598
4	25.5183	52.9772	110.906	3.86035	7.99497	16.5507
6	25.8105	52.9221	110.753	3.85167	8.02208	16.6074
8	16.4302	34.6895	70.6967	2.5807	5.087	10.4931
10	17.2411	33.7304	71.9502	2.46161	5.39557	11.9367
12	16.4415	33.758	72.6115	2.60143	5.38757	11.7147
14	18.1439	33.7759	71.2912	2.60458	5.39396	11.3473
16	11.9957	23.2279	49.0511	1.96937	3.87801	7.83966
18				1.85494	3.83097	7.94479

Tabla 4.9: Tiempos (en segundos) obtenidos para el código `quickHull`

4.2.11. Algoritmo `quickSort`

El conocido algoritmo de ordenación `quickSort` [83], hace uso de la recursividad para ordenar n números, empleando para ello un tiempo $O(n \log n)$. Presentamos este algoritmo como otro ejemplo de paralelismo multinivel a través del uso de recursividad.

El listado 4.17 recoge una versión esquemática del algoritmo, donde en la línea 11 se produce la llamada recursiva que realiza una división del vector actual en dos subvectores. Esta división no es simétrica, sino que depende de la posición del elemento pivote que se calcula a partir de la línea 6 (el código no se muestra por brevedad).

Dado que el bucle de la línea 10 del listado 4.17 produce tan sólo dos llamadas recursivas, estamos ante un caso idóneo para la aplicación de paralelismo anidado. El reducido número de tareas que se generan en cada nivel permite alcanzar una gran profundidad en los niveles de paralelismo que se obtienen, pudiendo representarse éste por un árbol binario. Por otro lado, la directiva de pesos de la línea 8 favorece que se asignen grupos de procesadores más grandes a aquellas tareas con una mayor estimación de carga computacional, lo que conlleva una mayor profundidad en el paralelismo multinivel y un mejor equilibrado de carga.

La figura 4.25 muestra la aceleración obtenida en el `Cray T3E` mediante el algoritmo `quickSort` para diferentes tamaños de vectores, mientras que la tabla 4.10 recoge los tiempos obtenidos. La aceleración se ve afectada

```
1  void qs(int *v, int first, int last) {
2    int size[2], start[2], end[2],
3                *res[2], pivot, i, temp;
4
5      /* calculating pivot... */
6      ...
7  #pragma omp parallel for
8  #pragma llc weight(size[i])
9  #pragma llc result(res[i], size[i])
10       for(i = 0; i < 2; i++) {
11         qs(v, start[i], end[i]);
12       }
13   }
14 }
```

Listado 4.17: Algoritmo `quickSort` mediante un bucle paralelo `llc`

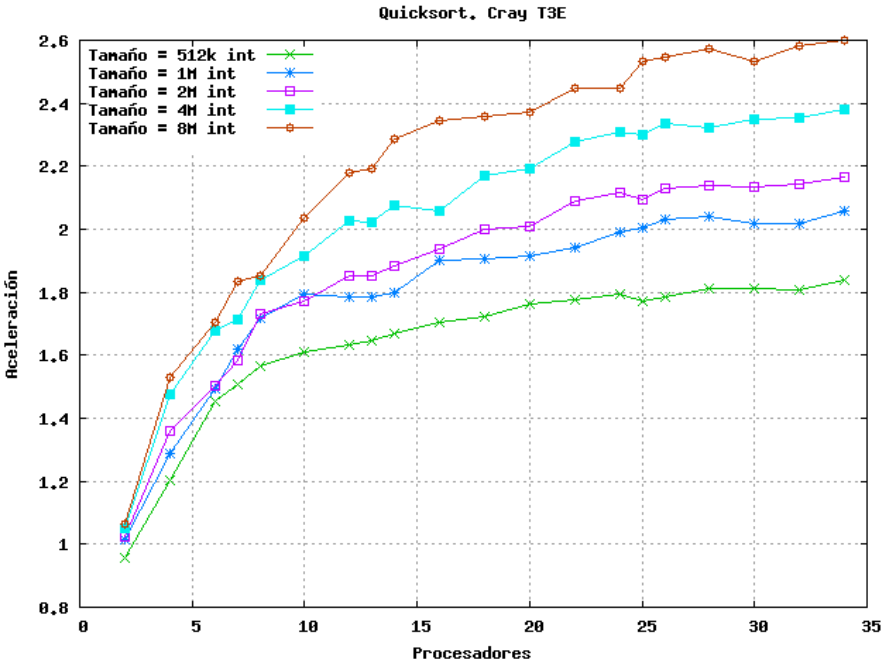

Figura 4.25: Aceleración para el código `quickSort` en el `Cray T3E`

directamente por el número de elementos a computar, y esto se debe a que el ancho de banda de entrada y salida del algoritmo es lineal con respecto a este tamaño, lo que constituye un límite inferior al tiempo de cualquier algoritmo de ordenación. Al ser el logaritmo del tamaño del vector de entrada un límite en la aceleración, la única forma de incrementar el rendimiento es aumentar el tamaño del vector. Sin embargo, en este caso se ha alcanzado el valor máximo de memoria física que el sistema nos permite usar, por lo cual la aceleración obtenida presenta unos valores relativamente limitados.

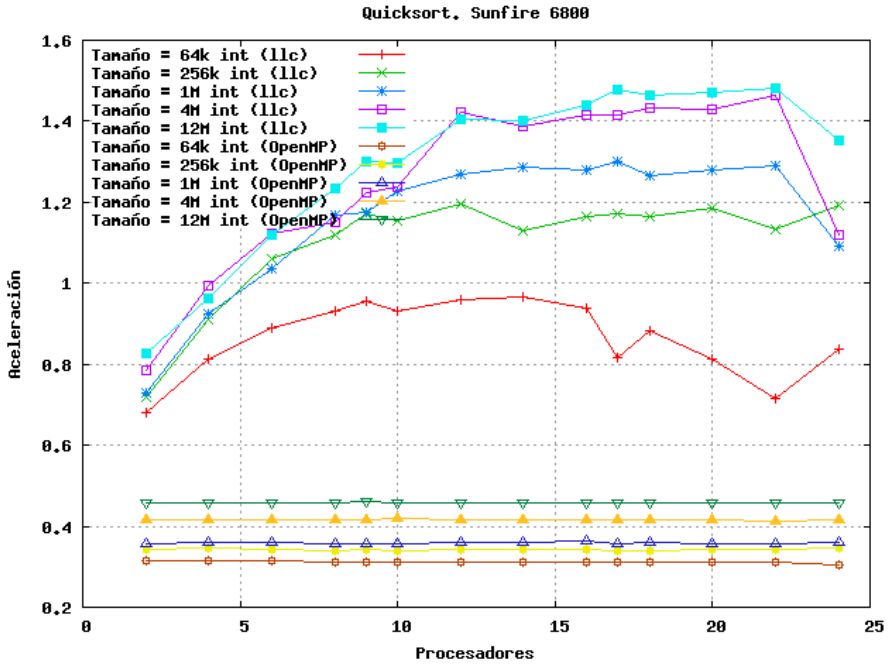

Figura 4.26: Aceleración pare el código `quickSort` en el `SunFire 6800`

Un caso extremo es el que se presenta en la figura 4.26 que recoge los resultados obtenidos en el `SunFire 6800`. Este sistema nos permite comparar la implementación de `llc` con `OpenMP`. Como se puede observar, el compilador de `OpenMP` no aplica paralelismo multinivel, sino que serializa las tareas anidadas obteniéndose resultados constantes independientemente del número de procesadores empelados. Dada la sobrecarga que produce el tratamiento de las directivas en las llamadas recursivas, los resultados de `OpenMP` son del orden de dos a tres veces peores que el equivalente secuencial. Por su parte, `llc` sí es capaz de obtener aceleración mediante el uso de paralelismo anidado, si bien el valor de esta aceleración es reducido.

	Cray T3E				
procs.	N = 512K	N = 1M	N = 2M	N = 4M	N = 8M
SEQ	0.445424	0.971324	2.04967	4.47062	9.10727
2	0.483351	0.9824	2.06681	4.43183	9.37173
4	0.374959	0.764298	1.5637	3.13603	6.32845
6	0.315691	0.675123	1.4291	2.7958	5.677
7	0.305843	0.614584	1.35477	2.72754	5.28594
8	0.29591	0.577123	1.19852	2.51431	5.24778
10	0.287904	0.553333	1.17218	2.41277	4.78703
12	0.283847	0.555026	1.12867	2.24712	4.42112
13	0.282636	0.555132	1.12459	2.25054	4.40355
14	0.278022	0.552988	1.11403	2.20307	4.28018
16	0.270095	0.522643	1.08327	2.22212	4.15757
18	0.266378	0.52179	1.05069	2.12126	4.12961
20	0.259307	0.51901	1.04795	2.10292	4.11117
22	0.257458	0.512443	1.00262	2.01757	4.00719
24	0.255028	0.497354	0.992238	1.99481	4.01202
25	0.258648	0.494526	1.0051	2.00246	3.85097
26	0.255981	0.487074	0.984527	1.96813	3.80394
28	0.253103	0.484535	0.980199	1.98754	3.7595
30	0.252162	0.489414	0.976877	1.96203	3.82033
32	0.252462	0.489203	0.976187	1.95094	3.7708
34	0.248174	0.48031	0.96559	1.93393	3.7386

Tabla 4.10: Tiempos (en segundos) obtenidos para el código `quickSort` en el `Cray T3E`

4.2.12. Algoritmo FFT D&C

En el apartado 4.2.4 (página 161) se estudió la FFT de "6 pasos" de Bailey. En este caso presentamos otro algoritmo que calcula la FFT [38], siguiendo esta vez un enfoque *divide y vencerás*.

El algoritmo se muestra en el listado 4.18 y recibe como entrada el puntero a a la señal de entrada, un vector W que contiene las raíces unitarias, el número de puntos de la señal (N), un valor entero stride que indica la distancia entre elementos que se debe tener en cuenta en la división de las fases del algoritmo y un puntero D auxiliar utilizado durante la fase de combinación. La función devuelve un puntero A a la señal transformada.

El listado 4.18 contiene la paralelización realizada con llc de la fase resolución de subproblemas. La línea 18 contiene la llamada a la propia función FFT() definida en la línea 1, estando esta llamada encerrada dentro del contexto del constructo paralelo de la línea 16, logrando de este modo paralelismo multinivel a través del uso de recursividad. En este caso, en cada nivel también se generan dos tareas, como indica el bucle de la línea 17 y, puesto que cada tarea procesa el mismo número de elementos, no existe ninguna asimetría en el peso de cada tarea, por lo que no se ha utilizado la directiva de equilibrado de carga.

El listado 4.19 conserva el paralelismo recursivo de la fase de resolución de subproblemas (directiva de la línea 4), pero se ha añadido un nuevo constructo en la línea 13 para paralelizar también la fase de combinación.

La figura 4.27 muestra los resultados obtenidos en Cray T3E para el código del listado 4.18 donde sólo se paralelizaba la fase de resolución de subproblemas para diferentes tamaños de problema. La gráfica presenta un comportamiento escalonado debido a las características del propio algoritmo, que necesita un número potencia de dos de procesadores para que sea posible aumentar la profundidad del paralelismo multinivel, lo cual produce los saltos de aceleración que pueden observarse al ejecutar el experimento con 2, 4, 8, 16 y 32 procesadores. Como puede observarse, los mejores resultados se obtienen al aumentar el tamaño del vector de entrada, ya que esto también aumenta el grano del problema.

La figura 4.28 presenta, a su vez, los resultados de la ejecución del código del listado 4.19 donde, además de la fase de resolución de subproblemas, también se paralelizaba la fase de combinación. Los resultados muestran un comportamiento similar al de la figura 4.27 anterior, si bien la aceleración total obtenida es inferior. Esto se debe a que al paralelizar la fase de

```
1  void FFT(Complex *A, Complex *a, Complex *W, unsigned N,
2               unsigned stride, Complex *D) {
3    Complex *B, *C, Aux, *pW;
4    unsigned i, n;
5
6    if(N == 1) {
7      A[0].re = a[0].re;
8      A[0].im = a[0].im;
9    }
10   else {
11     /* Division phase without copying input data */
12     n = (N>>1);
13
14     /* Subproblems resolution phase */
15 #pragma omp parallel for
16 #pragma llc result (D+i*n, n)
17     for (i = 0; i <= 1; i++) {
18       FFT(D+i*n, a+i*stride, W, n, stride<<1, A+i*n);
19     }
20
21     /* Combination phase */
22     ...
23   }
24 }
```

Listado 4.18: Paralelización de la fase de resolución de subproblemas del algoritmo `FFT` mediante un código `llc`

```
1       ...

3       /* Subproblems resolution phase */
4   #pragma omp parallel for
5   #pragma llc result (D+i*n, n)
6       for(i = 0; i <= 1; i++) {
7         FFT(D+i*n, a+i*stride, W, n, stride<<1, A+i*n);
8       }

10      /* Combination phase using all the group */
11      B = D;
12      C = D + n;
13  #pragma omp parallel for
14  #pragma llc result (A+i, 1, A+i+n, 1)
15      for(i = 0; i <= n-1; i++) {
16        pW = W + i * stride;
17        Aux.re = pW->re * C[i].re - pW->im * C[i].im;
18        Aux.im = pW->re * C[i].im + pW->im * C[i].re;

20        A[i].re = B[i].re + Aux.re;
21        A[i].im = B[i].im + Aux.im;
22        A[i+n].re = B[i].re - Aux.re;
23        A[i+n].im = B[i].im - Aux.im;
24      }
```

Listado 4.19: Paralelización de la fase de resolución de subproblemas y de combinación del algoritmo `FFT` mediante un código `llc`

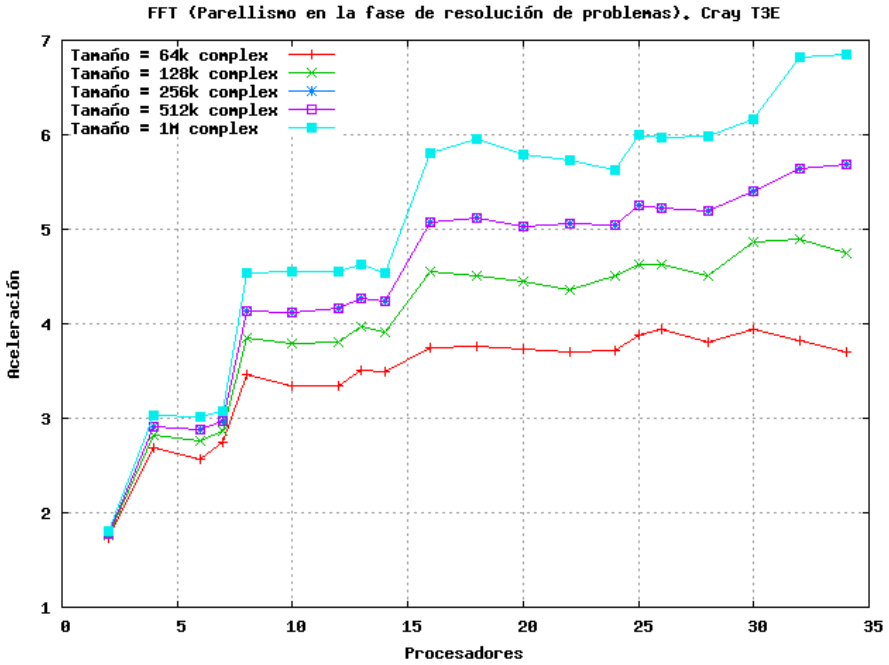

Figura 4.27: Aceleración para la paralelización de la fase de resolución de subproblemas para el algoritmo `FFT` en el `Cray T3E`

combinación, se ha disminuido el grano total de las tareas. Además, la gestión del paralelismo multinivel es menos eficiente: cada llamada recursiva a la función `FFT()` de la línea 7 apenas ejecutará ningún código, y sólo servirá para generar las tareas paralelas del bucle de la línea 15 que lleva a cabo la fase de combinación.

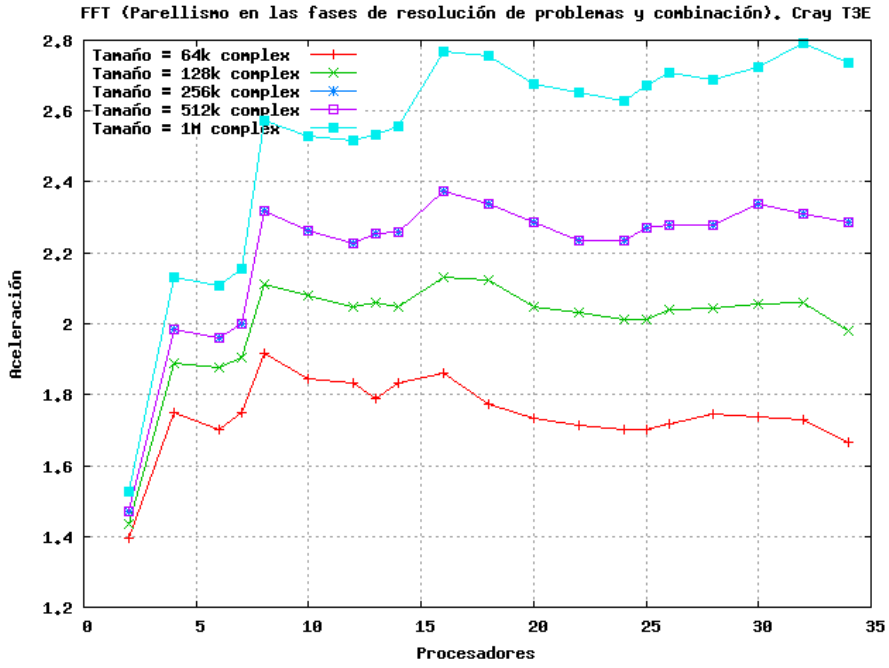

Figura 4.28: Aceleración para la paralelización de la fase de resolución de subproblemas y combinación para el algoritmo `FFT` en el `Cray T3E`

La tabla 4.11 recoge los tiempos a partir de los cuales se obtuvieron las figuras 4.27 y 4.28. Para una mejor presentación esta tabla sólo recoge el tamaño mínimo (N = 64K) y el máximo (N = 1M), estando los tiempos de los tamaños restantes comprendidos entre los límites que aquí se muestran.

4.2.13. Algoritmo `USMV`

`USMV` es el nombre con el se conoce a la operación de producto entre matriz dispersa y vector en *BLAS* (***B**asic **L**inear **A**lgebra **S**ubprograms*) [96].

El listado 4.20 muestra la paralelización del bucle principal del producto $y = y + \alpha Ax$, donde x e y son los vectores y A la matriz dispersa. Los elementos de la matriz son almacenados por filas, guardando también

	Cray T3E			
	Res. Subproblemas		*R. Subp. & Comb.*	
procs.	N = 64K	N = 1M	N = 64K	N = 1M
SEQ	0.280198	5.96322	0.279796	5.96429
2	0.161285	3.31262	0.200333	3.90837
4	0.104128	1.96483	0.159937	2.79517
6	0.109547	1.98147	0.164624	2.82899
7	0.101878	1.9377	0.160066	2.77111
8	0.080986	1.31553	0.146077	2.31824
10	0.083598	1.31145	0.151855	2.35731
12	0.083973	1.30913	0.152897	2.37069
13	0.079756	1.28683	0.156345	2.35631
14	0.080207	1.31288	0.152805	2.3297
16	0.074805	1.02563	0.150173	2.15527
18	0.07441	1.00269	0.158274	2.16259
20	0.075133	1.02928	0.161291	2.22968
22	0.075667	1.0401	0.16342	2.2472
24	0.075615	1.05945	0.164505	2.26989
25	0.072226	0.993985	0.164401	2.23041
26	0.071146	0.998542	0.162637	2.20151
28	0.073675	0.99802	0.160513	2.21982
30	0.071128	0.967681	0.16109	2.18975
32	0.073361	0.873524	0.161933	2.13616
34	0.075754	0.869233	0.167939	2.17999

Tabla 4.11: Tiempos (en segundos) obtenidos para los código `FFT D&C`

punteros al primer elemento de cada fila en el vector `ptr`. En este código, cada iteración del bucle externo (línea 2) realiza un producto entre la fila de la matriz dispersa y el vector x, produciendo un elemento del vector de solución y.

El código usa tres macros de C para acceder al índice de la fila (`index1_coordinate(ptr)`, linea 5), de la columna (`index2_coordinate(ptr)`, línea 8) y el valor del elemento de la matriz dispersa al que apunta `ptr` (`value_coordinate(ptr)`, línea 7). Una cuarta macro (`inc_coordinate(ptr)`, línea 9) mueve el puntero al siguiente elemento de la misma fila, mientas que `incx` e `incy` permiten el acceso a los vectores x e y usando desplazamientos no unitarios.

La paralelización de este código es inmediata al tener en cuenta que los diferentes productos son totalmente independientes, y se obtiene mediante la directiva paralela de la línea 1 del listado 4.20. Sin embargo, el acceso a memoria que se produce al escribir el resultado en el vector `y` (línea 12) sigue un patrón de acceso que no se conoce a priori, puesto que el índice se calcula en base a la variable `k`, cuyo valor no puede determinarse hasta la ejecución de cada iteración. Por este motivo, no es posible utilizar la directiva de comunicación de resultados `result` como en las paralelizaciones previamente estudiadas, teniendo que usarse las directivas específicas de `llc` que gestionan los accesos no contiguos a memoria.

Como se explicó en el capítulo 2, `llc` permite dos tipos de accesos no contiguos a memoria en los bucles paralelos, el *acceso no contiguo con patrón regular* y el *acceso con patrón desconocido*. El primero de ellos se lleva a cabo mediante la directiva `llc rnc_result` y será ejemplificado en el apartado 4.5.1 (página 221), que trata este tipo de accesos mediante una directiva equivalente usada en el paralelismo de cola de tareas. Mientras, este listado 4.20 ilustra la gestión de los accesos a memoria con un patrón no conocido, a través de la directiva `rn_result` (línea 11).

Nótese que esta directiva, a diferencia de la mayoría de directivas de `llc`, ha de usarse dentro del bloque de código, inmediatamente antes de producirse el acceso para asegurar que las variables que intervienen en ella no cambian sus valores y los punteros delimitan correctamente la zona de memoria que se va a modificar.

La figura 4.29 presenta los resultados de `llc` obtenidos para este algoritmo en la `SGI O3000`, con tamaños de problemas de $N = M = 20000$ y 30000, y factores de dispersión del 1 % y 2 %. Como se puede observar, las curvas son similares, presentando un comportamiento que tiende a la linealidad, donde

```
#pragma omp parallel for private (ptr, temp, k, j)
  for (i = 0; i < Blks->size1; i++) {
    ptr = Blks->ptr[i];
    temp = 0.0;
    k = index1_coordinate (ptr); // 1st element in i-th row
    for (j=0; j < elements_in_vector_coord(Blks, i); j++) {
      temp += value_coordinate (ptr)
                * x[index2_coordinate(ptr) * incx];
      inc_coordinate (ptr);
  }
#pragma llc nc_result (&y[k * incy], 1, y)
  y[k * incy] += alpha * temp;
}
```

Listado 4.20: Bucle paralelizado en el algoritmo `USMV`

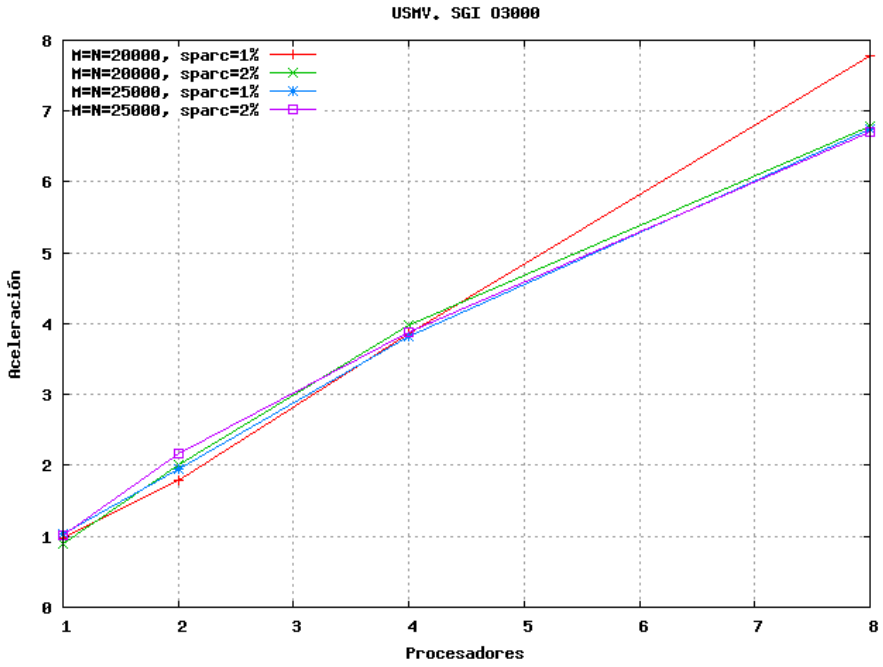

Figura 4.29: Aceleración para el código `USMV` en la `SGI O3000`

destaca el rendimiento obtenido con el menor tamaño e índice de dispersión.

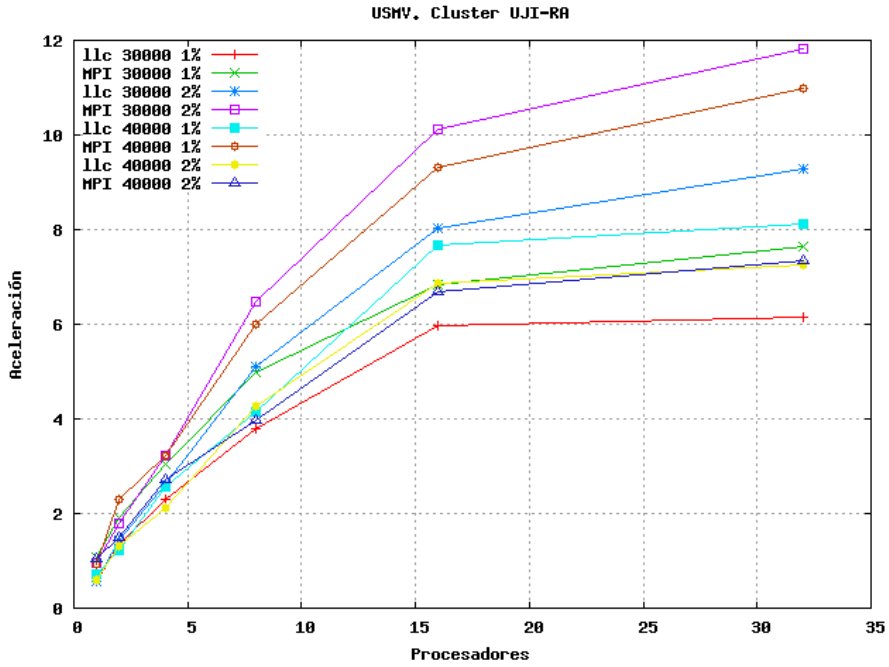

Figura 4.30: Aceleración para el código `USMV` en el `Cluster UJI-RA`

Por otro lado, la figura 4.30 compara en el `Cluster UJI-RA` los resultados obtenidos para las versiones de `llc` y *ad hoc* `MPI` con tamaños de de $N = M = 30000$ y 40000 y factores de dispersión del 1 % y 2 %. Los resultados obtenidos con `MPI` en este caso particular son sensiblemente mejores que los de `llc`. Esto se debe principalmente a dos factores: en primer lugar, el grano fino de los cálculos no favorece a las implementaciones en `llc`, cuyo código generado de comunicaciones se comporta mejor cuanto más grande sea el grano de las operaciones. Por otro lado, es de esperar que al usar la directiva de comunicación de resultados con acceso aleatorio se obtenga un menor rendimiento, debido al sobrecosto generado de la gestión de las listas de regiones a comunicar, como se explicó en el apartado 3.6.3 (página 141).

4.2.14. Algoritmo `Dinámica Molecular` (con directiva `nc_result`)

El algoritmo de dinámica molecular ya ha sido presentado en el apartado 4.2.3 (página 159). Sin embargo, aquí mostramos una variante de este algoritmo que hemos desarrollado con el objetivo de estudiar el rendimiento

de la directiva `nc_result` en comparación con la directiva de comunicación de regiones contiguas `result`.

La directiva `nc_result` presenta el mayor coste dentro de todas las directivas de comunicación de `llc` en cuanto se refiere a la sobrecarga introducida. Esto se debe a que es necesario gestionar una lista de regiones de memoria a comunicar que es compactada mediante una heurística, con el objetivo de optimizar las comunicaciones reduciendo el número de las mismas y evitando en lo posible solapamiento de regiones.

```
1   pot = 0.0;
2   kin = 0.0;

4  #pragma omp parallel for default(shared)
5        private(i,j,k,rij,d) reduction(+ : pot, kin)
6  #pragma llc reduction_type (real8, real8)
7   for (i = 0; i < np; i++) {

9     /* compute potential energy and forces */
10 #pragma llc nc_result (&f[i], 1, f[0])
11    for (j = 0; j < nd; j++)
12      f[i][j] = 0.0;

14    for (j = 0; j < np; j++) {
15      if (i != j) {
16        d = dist(nd,box,pos[i],pos[j],rij);
17        pot = pot + 0.5 * v(d);
18        for (k = 0; k < nd; k++) {
19          f[i][k] = f[i][k] - rij[k]* dv(d) /d;
20        }
21      }
22    }
23    /* compute kinetic energy */
24    kin = kin + dotr8(nd,vel[i],vel[j]);
25  }
26  kin = kin*0.5*mass;
```

Listado 4.21: Bucle paralelizado con directiva `nc_result` en el algoritmo de Dinámica Molecular

En el estudio anterior de este algoritmo se presentó el código que calculaba la posición, velocidad y aceleración de cada partícula (listado 4.4, página 159). En esta ocasión haremos uso del listado 4.21 para presentar

la paralelización del bucle que calcula las fuerzas y energías (línea 7). La paralelización se lleva a cabo mediante la directiva de la línea 4 (continúa en línea 5) cuya cláusula de reducción se ve complementada por la directiva `reduction` de `llc` en la línea 6.

A pesar de que los accesos a memoria son contiguos, mediante la directiva `nc_result` de la línea 10 forzamos a `llc` a que los gestione como si se tratase de accesos *aleatorios*. Esto nos permitirá tener una medida de la sobrecarga introducida por la gestión de las regiones de memoria y la heurística de compactación, ya que podremos comparar los resultados con una versión de este algoritmo que utiliza la directiva de acceso contiguo, así como otra versión *ad hoc* realizada en `MPI`.

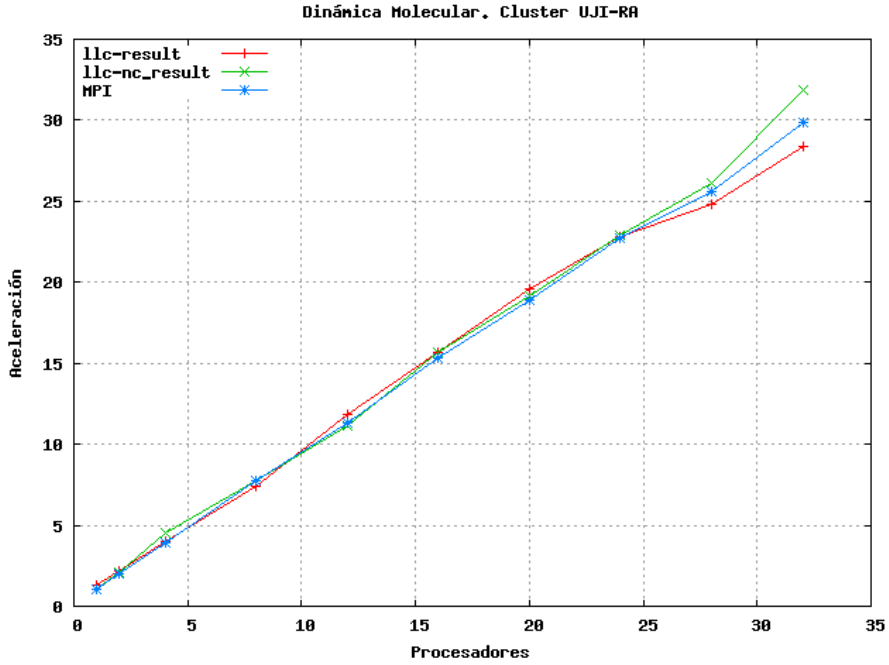

Figura 4.31: Aceleración para el código Dinámica Molecular con directiva `nc_result` en el `Cluster UJI-RA`

Los resultados computacionales obtenidos en el `Cluster UJI-RA` [50, 52] se presentan en la figura 4.31. Esta figura muestra como, para este ejemplo, no existen diferencias notables entre las tres versiones, lo que demuestra que la gestión de las regiones de memoria es eficiente para aquellos casos en los que se reciben regiones contiguas, sin diferencias apreciables sobre el equivalente resultante de tratar esas regiones como contiguas mediante la directiva específica.

4.3. Secciones paralelas

Las *secciones paralelas* son otro de los métodos usados para explotar paralelismo en `OpenMP`. Este paradigma ha sido implementando en `llc` con el fin de asegurar la compatibilidad con `OpenMP`, si bien no es una aproximación generalmente usada para obtener paralelismo, al menos en el ámbito de códigos de interés científicos con los que usualmente trabajamos.

Para suplir esta carencia, haremos uso de dos de los códigos ya estudiados en los bucles paralelos, usando en esta ocasión secciones para llevar a cabo la paralelización.

4.3.1. Algoritmo `quickHull` (con directiva `sections`)

El algoritmo `quickHull` [20] mediante el que se obtiene la envolvente convexa de un conjunto de puntos ya ha sido estudiado en el subapartado 4.2.10 (página 192). En este caso presentaremos una paralización mediante el uso de secciones de código que serán ejecutadas en paralelo.

El listado 4.22 recoge esta paralelización, observándose en la línea 8 la directiva `sections` que abre la región donde se localizan las secciones paralelas, declaradas mediante la directiva `section` en las líneas 10 y 13. Cada una de estas secciones contiene la llamada recursiva a la función `sections_q_hull()` que gestiona el particionado del conjunto de puntos actual en dos nuevos subconjuntos que serán computados en paralelo, generando de esta forma un árbol binario de divisiones. Este particionado finaliza cuando el grupo de procesadores actual es unitario, caso en el que se invoca la versión secuencial del algoritmo.

Como se estudió en el apartado 3.5.3 (página 132), la traducción de las secciones paralelas se lleva a cabo mediante una primera etapa de conversión. En este apartado estamos interesados en comparar el rendimiento de las secciones frente a los bucles paralelos, para obtener una estimación de la posible pérdida de rendimiento que el proceso de conversión puede introducir.

Para ello, los resultados computacionales de la versión con secciones fueron obtenidos conjuntamente con los de los bucles paralelos que se muestran en la figura 4.23 (página 195), con nubes de 2, 4 y 8 $\times 2^{20}$ puntos y ejecutando el algoritmo tres veces para obtener los tiempos medios y reducir el efecto de fluctuaciones en las ejecuciones.

La figura 4.32 muestra estos resultados, en los que hemos establecido

```
1  void sections_q_hull(intType **f_hull, intType **l_hull,
       intType *m,
2                       intType * size) {

4    ...
5    if (max != NULL){
6      partition(firstA,lastB,max,&lastA,&firstB,&maxA,&maxB);

8  #pragma omp parallel sections
9  {
10 #pragma omp section
11 #pragma llc result (&sizeA, 1)
12       sections_q_hull(&firstA, &lastA, maxA, &sizeA);
13 #pragma omp section
14 #pragma llc result (&sizeB, 1)
15       sections_q_hull(&firstB, &lastB, maxB, &sizeB);
16 }
17   ...
18   }
19 }
```

Listado 4.22: Paralelización del algoritmo `quickHull` implementado usando `sections`

como referencia la aceleración obtenida con la versión de bucles paralelos (aceleración = 1), normalizando la aceleración de las secciones ($a_{sect-norm} = \frac{a_{sect}}{a_{for}}$). En esta figura podemos observar como la diferencia de rendimiento de ambas versiones es poco significativa.

Para algunos valores, incluso el rendimiento de la paralelización realizada con secciones supera al de los bucles paralelos. Este hecho se debe a que, mientras que en la paralelización con secciones se indican directamente las variables en las llamadas recursivas (líneas 12 y 15 del listado 4.22 de la página 212), para el caso de bucles paralelos es necesario definir una serie de vectores auxiliares que simulan este comportamiento para poder aplicar la paralelización (línea 11 del listado 4.16 de la página 194). La gestión de estos vectores y el mayor costo de los accesos influyen en la diferencia de rendimiento obtenida. El lector puede consultar los tiempos obtenidos por cada una de las versiones para los distintos tamaños en la tabla 4.12.

La figura 4.33 presenta la aceleración normalizada de la versión `sections` obtenida tomando como referencia los resultados de los bucles paralelos

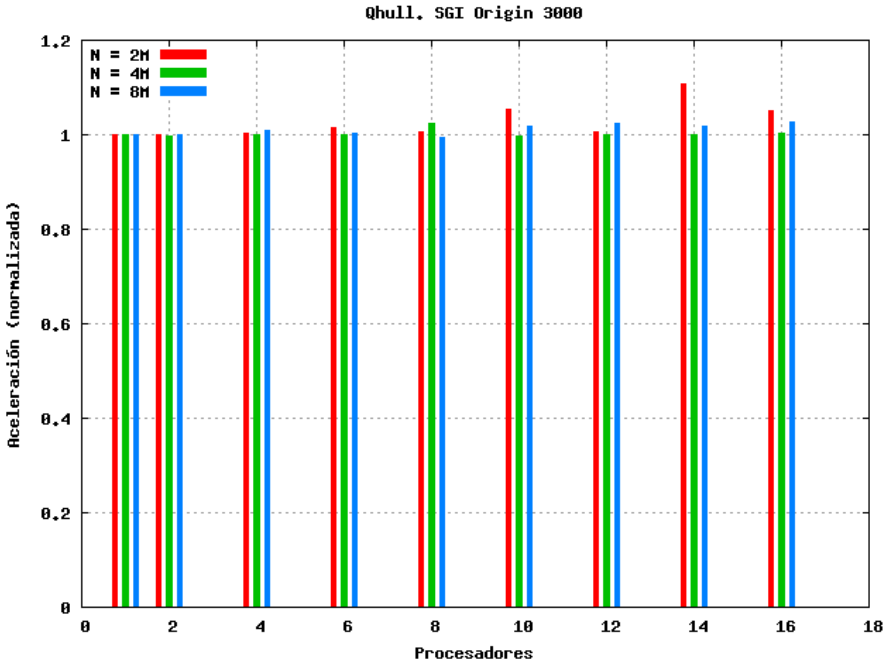

Figura 4.32: Aceleración normalizada para el código `quickHull` implementado usando `sections` en la `SGI O3000`

mostrados en la figura 4.24 (página 195). Ambas ejecuciones fueron realizadas bajo las mismas condiciones y usando los mismos parámetros: 2, 4 y 8 $\times 2^{20}$ puntos y tres repeticiones. Los resultados de esta figura se asemejan a los discutidos anteriormente y confirman las observaciones realizadas.

4.3.2. Algoritmo `quickSort` (con directiva `sections`)

El algoritmo de ordenación `quickSort` [83] paralelizado mediante bucles ya ha sido presentado en el subapartado 4.2.11 (página 196). El listado 4.23 muestra una paralelización de este algoritmo usando secciones paralelas.

En la línea 8 se crea una región paralela donde se localizan las secciones. La primera sección está formada por la llamada recursiva a la función de ordenación de las líneas 13 y 14, utilizando para ello la directiva `section` de la línea 10 y la comunicación de resultados de la línea 12. Además, se ha añadido una directiva para equilibrado de la carga en la línea 11, usando el tamaño del vector como peso de la tarea asignada. El resto de código recoge la especificación de la segunda sección, que corresponde con la segunda parte

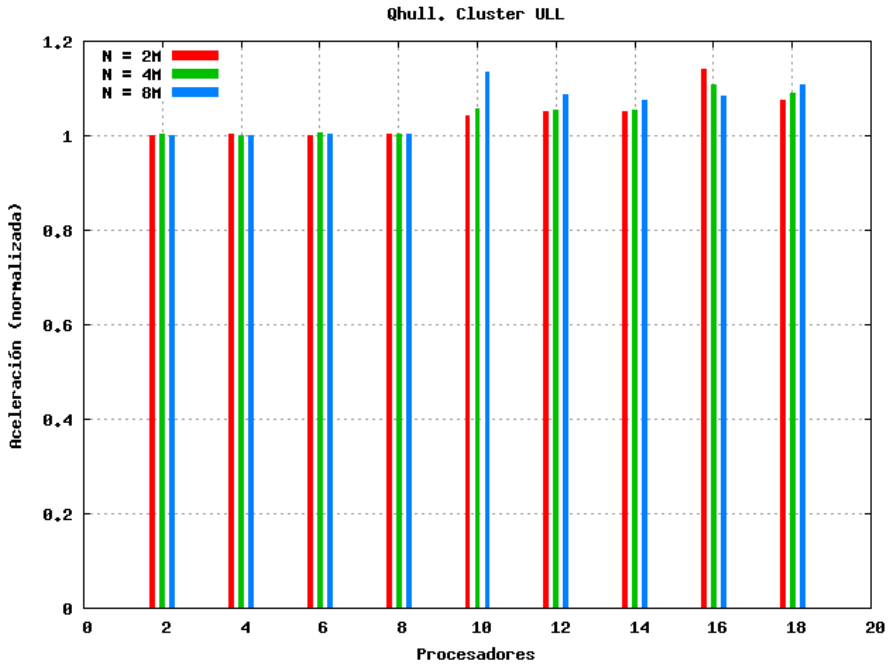

Figura 4.33: Aceleración normalizada para el código `quickHull` implementado usando `sections` en el `Cluster ULL`

	SGI O3000					
	N = 2M		N = 4M		N = 8M	
procs.	`sect`	`for`	`sect`	`for`	`sect`	`for`
SEQ	64.5566		134.663		281.979	
2	33.6785	33.6759	70.3785	70.269	146.71	146.913
4	25.423	25.5183	52.9228	52.9772	109.978	110.906
6	25.4336	25.8105	52.9348	52.9221	110.578	110.753
8	16.342	16.4302	33.9192	34.6895	71.0901	70.6967
10	16.3439	17.2411	33.7884	33.7304	70.6267	71.9502
12	16.3658	16.4415	33.7504	33.758	71.0	72.6115
14	16.3674	18.1439	33.7363	33.7759	70.1357	71.2912
16	11.4288	11.9957	23.1628	23.2279	47.7907	49.0511

Tabla 4.12: Tiempos (en segundos) obtenidos para el código `quickHull` comparando versiones con directivas `sections` y `for`

```
1  void qs (int *v, int first, int last) {
2    int pivot, temp, i, j;
3
4    pivot = v[(first + last) / 2];
5    /* Cálculo de los límites i y j */
6    ...
7
8  #pragma omp parallel sections
9    {
10 #pragma omp section
11 #pragma llc weight(j - first + 1)
12 #pragma llc result(v + first, (j - first + 1))
13   if (first < j)
14     qs(v, first, j);
15 #pragma omp section
16 #pragma llc weight(last - i + 1)
17 #pragma llc result(v + i, (last - i + 1))
18   if (i < last)
19     qs(v, i, last);
20   }
21 }
```

Listado 4.23: Paralelización del algoritmo `quickSort` implementado usando `sections`

del vector a ordenar, mediante el uso de las mismas directivas.

Al igual que para el algoritmo `quickHull`, en este apartado estamos más interesados en realizar un estudio del comportamiento de la traducción de las secciones paralelas que en medir el rendimiento del algoritmo `quickSort`, del que ya conocemos que ofrece unos resultados pobres según se presentó en el subapartado 4.2.11 (página 196). Por ello realizaremos una comparación de los resultados entre la aproximación de bucles y secciones paralelas aplicadas sobre el mismo algoritmo.

La figura 4.34 muestra la comparativa de la ejecución de las paralelizaciones con bucles y secciones paralelas en el `Cluster ULL`, estableciendo como referencia la aceleración obtenida con la versión de bucles paralelos (aceleración = 1) y normalizando la aceleración de las secciones. El vector de entrada para estos resultados es de 20 $\times 2^{20}$ enteros y se realizan tres repeticiones, obteniendo la media de los tiempos para reducir las fluctuaciones.

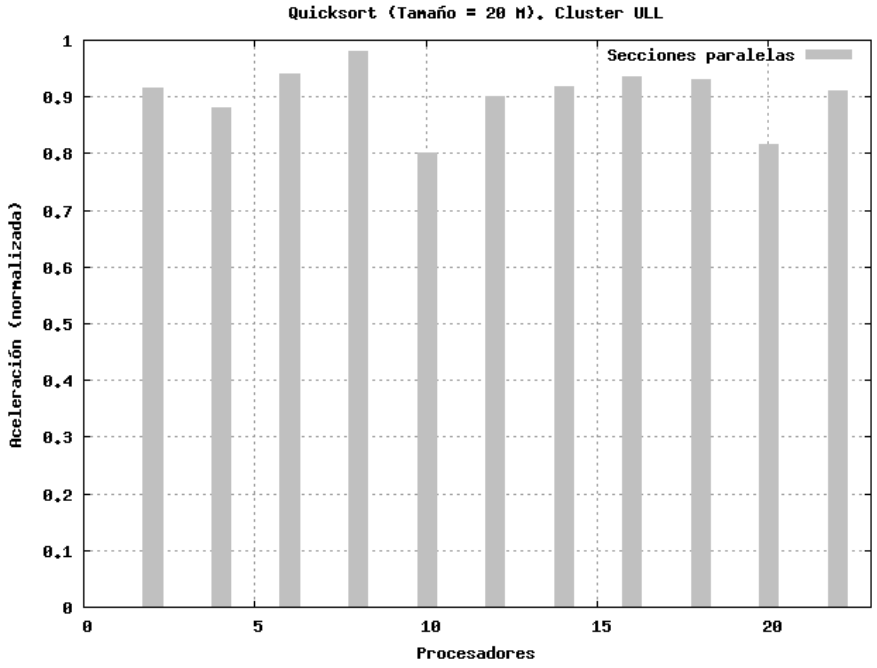

Figura 4.34: Aceleración normalizada para el código `quickSort` implementado usando `sections` en el `Cluster ULL`

Como se puede apreciar en esta figura 4.34, el rendimiento de ambas aproximaciones ofrece resultados similares, si bien para algunas ejecuciones de este algoritmo los valores de las secciones paralelas muestran unas pérdidas con respecto a los bucles, siendo el valor medio de un 10 %. Esta diferencia de rendimiento se produce por el sobrecosto del código de conversión de secciones a bucles que introduce el compilador, y consideramos que queda justificada por una mayor simplicidad en el diseño del compilador, la reutilización de código y el escaso uso real que hemos encontrado de las secciones como paradigma de paralelización en códigos científicos.

La figura 4.35 presenta los resultados obtenidos para el mismo experimento en la `SGI O3000`. En este caso el rendimiento de las secciones paralelas presenta una pérdida ligeramente mayor que en el caso anterior, entre un 75 % y el 100 % del de los bucles, siendo el valor medio de la pérdida de un 15 %.

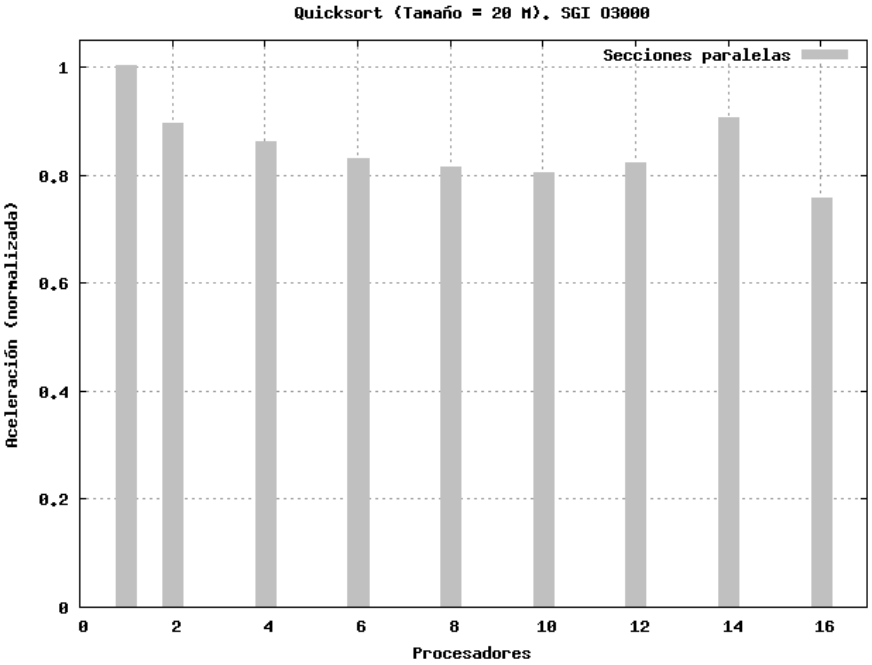

Figura 4.35: Aceleración normalizada para el código `quickSort` implementado usando `sections` en la `SGI O3000`

4.4. Segmentación paralela

Otro de los paradigmas de extracción de paralelismo que se ha implementado en `llc` es la *segmentación paralela* o *pipeline*. De este modo, se proporciona al desarrollador la posibilidad de transformar aquellos algoritmos que puedan ser modelados mediante el uso de un pipeline en aplicaciones paralelas donde las distintas etapas son ejecutadas en paralelo.

4.4.1. Algoritmo del `Problema de Asignación de Recursos`

El *SRAP* (***S**ingle **R**esource **A**llocation **P**roblem*) [111] es un problema de optimización que trata de encontrar la asignación óptima de M unidades de un recurso indivisible entre N tareas demandantes. La función $f_n(r)$ devuelve el beneficio obtenido por la asignación de r unidades del recurso a la tarea n-ésima. A través de programación dinámica, el problema `SRAP` puede ser reducido al cálculo de $G_{N-1}(M)$ usando las ecuaciones:

$$G_n(r) = \text{máx}\{G_{n-1}(r-i) + f_n(i) : \quad i \in \{0 \ldots r\}\}$$

En el listado 4.24 se muestra el código de una función que toma como parámetros `N`, `M` y la función de beneficio `f(n, r)`. Esta función rellena las tablas `G[n][r]` y `L[n][r]` con los mejores beneficios y las decisiones óptimas, respectivamente. Para resolver este problema, el algoritmo del listado implementa un paradigma de segmentación mediante la directiva `llc pipeline` de la línea 4, que indica que las N iteraciones del bucle de la línea 6 sean organizadas de forma que cada iteración corresponda con una etapa de este pipeline, mientras que la directiva `result` de la línea 5 garantiza que al finalizar el pipeline los resultados sean distribuidos entre los distintos grupos de procesadores.

La etapa inicial de este pipeline (tarea `n==0`) se localiza entre las líneas 7 a 11. La directiva `send` de la línea 10 inserta un elemento en el flujo de datos de la etapa siguiente. A continuación se encuentra el código de las restantes etapas, donde, en la línea 14, la directiva `receive` obtiene los valores de la etapa previa, calcula el nuevo valor `G[n][r]` y lo envía a la siguiente etapa mediante la directiva `send` de la línea 25, hasta que que la n-ésima iteración rellena la n-ésima columna de las matrices `G` y `L`.

La figura 4.36 muestra la aceleración obtenida por este algoritmo en diversos sistemas [56, 60] para un total de 350 tareas y 4000 recursos, excepto en `SGI O3000` donde el número de tareas es 150 y el de recursos 2000. Como puede apreciarse, los resultados muestran un comportamiento similar, tendiendo la curva a tener un comportamiento lineal donde la pendiente depende de la máquina y tamaño del problema donde fue ejecutada, habiéndose obtenido el mejor resultado en el `Cluster UJI-RA`. Los datos de tiempo a partir de los cuales se han obtenido las aceleraciones de la figura 4.36 pueden consultarse en la tabla 4.13.

4.5. Paralelismo de cola de tareas

El modelo de paralelismo de colas de tareas se aplica principalmente en aquellas situaciones donde no es posible aplicar otros paradigmas, como bucles sin índice o con condición de parada, y su uso se aconseja donde exista un desequilibrio en las cargas de las tareas, ya que el propio modelo lleva implícito el equilibrado de las mismas.

El paralelismo de colas tareas se realiza en `llc` mediante la directiva `taskq` y se ha seguido una sintaxis similar a la propuesta por `Intel-KAI`

```
1  int srap(int N, int M, profit f, table G, table L) {
2    int r, n, i, decision_i, temp, pos;
3
4  #pragma llc parallel pipeline
5  #pragma llc result (&G[n][0], M+1, &L[n][0], M+1)
6    for(n = 0; n < N; n++) {
7      if(n == 0)
8        for(r = 0; r <= M; r++) {
9          G[0][r] = f(0,r);
10 #pragma llc send (&G[0][r], 1)
11       }
12     else
13       for(r = 0; r <= M; r++) {
14 #pragma llc receive (&G[n-1][r], 1)
15         temp = G[n-1][r];
16         pos = 0;
17         for(i = 1; i <= r; i++) {
18           decision_i = G[n-1][r-i] + f(n, i);
19           if(decision_i > temp)  {
20             temp = decision_i;
21             pos = i;
22           }
23         }
24         G[n][r] = temp;
25 #pragma llc send (&G[n][r], 1)
26         L[n][r] = pos;
27       }
28     }
29   return G[N-1][M];
30 }
```

Listado 4.24: Paralelización del algoritmo `SRAP` implementado usando `pipeline`

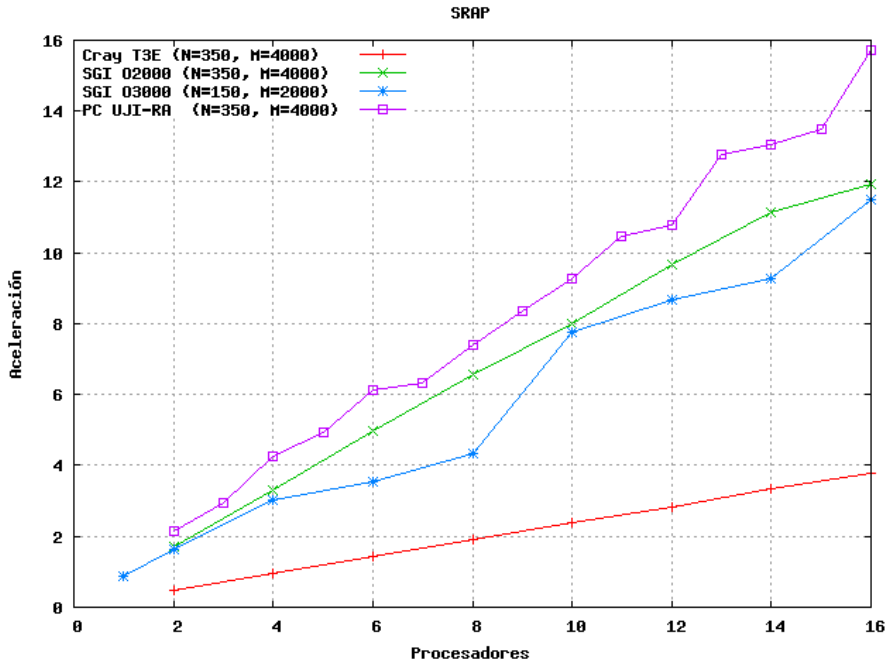

Figura 4.36: Aceleración para el código `SRAP` implementado usando `pipeline` en diversas plataformas

procs.	SGI O2000	SGI O3000	UJI-RA
SEQ	673.560	6.99912	36.4727
2	414.291	4.3013	18.5718
4	205.006	2.3022	9.36205
6	135.707	1.98374	6.31007
8	102.568	1.64337	5.03978
10	84.0139	0.90133	3.67283
12	69.8722	0.80552	3.24231
14	60.3588	0.75315	2.69088
16	56.2565	0.60717	2.38576

Tabla 4.13: Tiempos (en segundos) obtenidos para el código `SRAP`

[138]. Al no ser estas directivas un estándar y sólo estar implementados en el compilador `Intel` de `OpenMP`, no ha sido posible encontrar muchos códigos con los que comparar nuestros resultados.

Entre los códigos que han sido posible paralelizar mediante este modelo se encuentran los ejemplos proporcionados por el propio grupo `Intel-KAI`, algoritmos paralelizados con otros paradigmas en `llc` y que han sido adaptados al modelo de tareas y unos cuantos códigos de operaciones de álgebra matricial que están siendo paralelizados con estas directivas por grupos de investigación con los que mantenemos contacto.

4.5.1. Algoritmo de `Strassen`

El código que se muestra en el listado 4.25 corresponde con un porción del código del algoritmo paralelo de *Strassen* para la multiplicación de matrices [40]. Este código presenta la paralelización realizada por `llc`, tomando como código de partida la paralelización realizada por `Intel-KAI` [138], que a su vez se basa en [69].

Este código es propuesto por el grupo `Intel-KAI` como ejemplo de paralelización de una aplicación mediante el uso de sus directivas `taskq`, junto con otros códigos [145] que también han sido paralelizados con `llc`: cálculo de las *n-reinas*, serie de *fibonacci*, cálculo de *permanentes*, algoritmo de ordenación *multisort*, etc.

El algoritmo de `Strassen` contiene diversas áreas donde puede ser usado el constructo `taskq`. El listado 4.25 muestra una de estas regiones, donde la matriz original se divide en varias matrices más pequeñas y éstas son multiplicadas en paralelo usando para ello el constructo `taskq`. Un número mínimo de directivas `llc` deben ser añadidas sobre el código original `OpenMP` de `Intel-KAI`.

Estas directivas `llc` indican qué regiones de memoria deben ser comunicadas a otros grupos de procesadores al finalizar las tareas. De este modo, en la línea 7 (directiva `task_slave_data`) se indica que el esclavo que realice esta tarea debe devolver como resultado la matriz `tmp`.

Directivas similares a ésta se encuentran en las líneas 13, 20 y 27 (directiva `task_slave_rnc_data`), pero en este caso se trata de regiones de memoria no contiguas que siguen un patrón regular, y que describen qué regiones de la matriz `C` debe devolver cada esclavo. Por último, la directiva `taskq_master_result` de la línea 3 tiene como función que el maestro, único

```
1  ...
2  #pragma intel omp taskq
3  #pragma llc taskq_master_result (&C[0], n * n)
4  {
5    /* compute tmp <= a12 x b21 */
6    #pragma intel omp task
7    #pragma llc task_slave_data (&(tmp[0]), (ss * ss))
8    matrix_mult(ss, bs, ss, A, 0, ss, size,
9                B, ss, 0, size, tmp, 0, 0, ss, 1);

11   /* compute c12 <= (a11,a12) x (b12,b22)T */
12   #pragma intel omp task
13   #pragma llc task_slave_rnc_data (&(C[ss]), bs, \
14                                    (size - bs), ss)
15   matrix_mult(ss, n, bs, A, 0, 0, size,
16               B, 0, ss, size, C, 0, ss, size, 1);

18   /* compute c21 <= (a21,a22) x (b11,b21)T */
19   #pragma intel omp task
20   #pragma llc task_slave_rnc_data (&(C[ss*size]), \
21                                    ss, (size - ss), bs)
22   matrix_mult(bs, n, ss, A, ss, 0, size,
23               B, 0, 0, size, C, ss, 0, size, 1);

25   /* compute c22 <= (a21,a22) x (b12,b22)T */
26   #pragma intel omp task
27   #pragma llc task_slave_rnc_data (&(C[(ss*size)+ss]), \
28                                    bs, (size - bs), bs)
29   matrix_mult(bs, n, bs, A, ss, 0, size,
30               B, 0, ss, size, C, ss, ss, size, 1);
31 }
32 ...
```

Listado 4.25: Paralelización del algoritmo `strassen` implementado usando `taskq`

procesador que al finalizar dispondrá del resultado total, comunique éste al resto de procesadores del grupo.

Dadas las características de nuestra implementación del constructo `taskq`, en las ejecuciones se destina uno de los procesadores a las labores del master, creando y gestionando las tareas, enviando los subproblemas a los esclavos y recibiendo y combinando luego los resultados parciales. Esto supone la pérdida de un procesador efectivo para la ejecución de las tareas, por lo que la máxima aceleración que se podrá lograr es de $NP - 1$, siendo NP el número de procesadores totales del grupo. De este modo, la aceleración sólo se obtendrá a partir de tres procesadores, de forma que uno de ellos sea el maestro y los dos restantes se destinen como esclavos (si se utilizan 2 o menos procesadores, el código se secuencializa para evitar la sobrecarga de la gestión de la cola de tareas).

Este hecho queda patente en la figura 4.37 que recoge los resultados computacionales en diversas plataformas y tamaños de problemas [60]. En esta figura se aprecia cómo la aceleración para dos procesadores es de 1 (la ejecución se ha desarrollado secuencialmente), mientras que a partir de tres procesadores se observa como la curva empieza a crecer. Los tiempos empleados para obtener estas aceleraciones se muestran en la tabla 4.14.

La figura 4.37 también pone de manifiesto una característica intrínseca de la versión del algoritmo de `Straseen` paralelizado, donde el número máximo de matrices que se generan simultáneamente es de siete. Al corresponder cada tarea con la multiplicación de una de estas submatrices, el número de tareas y, por consiguiente la aceleración máxima, estarán limitadas a este valor, ya que `llc` no admite recursividad ni anidamiento en las tareas paralelas. En la figura 4.37 puede observarse como a partir de ocho procesadores (siete procesadores esclavos efectivos) la aceleración permanece prácticamente constante, debido a que se ha alcanzado el número máximo de tareas paralelas. Aumentar el número de procesadores entonces no supondrá alcanzar mayores cotas de aceleración.

4.5.2. Algoritmo `Mandelbrot` (con directiva `taskq`)

El algoritmo que calcula el conjunto de Mandelbrot [107] frecuentemente es un caso de estudio en cursos introductorios de CAP, ya que sirve de ejemplo para modelar situaciones en las que se produce desequilibrio de carga de trabajo.

La paralelización del código `Mandelbrot` mediante un bucle `for` paralelo

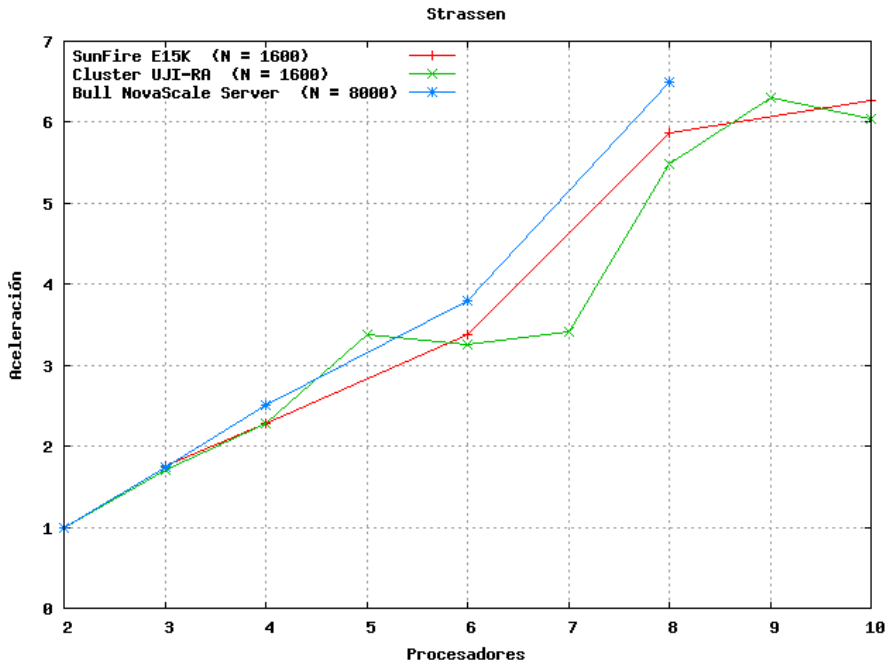

Figura 4.37: Aceleración para el código `Strassen` implementado usando `taskq`

procs.	SF E15K	UJI-RA	BULL
SEQ	44.670	24.9430	1053.510
2		24.9806	977.346
3	25.436	14.5477	566.818
4	19.492	10.9295	422.721
5		7.4099	394.909
6	13.219	7.6393	293.449
7		7.3077	291.454
8	7.600	4.54502	173.577
9		3.95256	
10	7.160	4.1311	

Tabla 4.14: Tiempos (en segundos) obtenidos para el código `strassen`

ya fue estudiada en el apartado 4.2.7 (página 174). Aquí presentaremos la paralelización de este algoritmo con un enfoque de cola de tareas, con el objetivo de reducir la pérdida de rendimiento de otros paradigmas cuando existe asimetría en las tareas. La paralelización resultante se muestra en el listado 4.26.

Como ya se estudió, para determinar si un punto C pertenece o no al conjunto de Mandelbrot, se aplica repetidamente la ley de recursión $Z_n = {Z_{n-1}}^2 + C$, aprovechando la propiedad de que los puntos externos al conjunto de Mandelbrot escapan al infinito (cuanto más lejos del conjunto, más rápido escapan), mientras que los puntos internos al conjunto nunca escapan al infinito.

De este modo, el número de iteraciones que es necesario aplicar para determinar si el punto permanece cerca del origen (convergencia) o escapa de él sin límite (divergencia) es el criterio de pertenencia de dicho punto al conjunto, a la vez que este número de iteraciones ofrece una estimación de la distancia de dicho punto al origen. Si se representa esta distancia asignando un color a un rango de direcciones, se obtiene como resultado el famoso fractal del conjunto Mandelbrot [68], que se muestra en la figura 4.38.

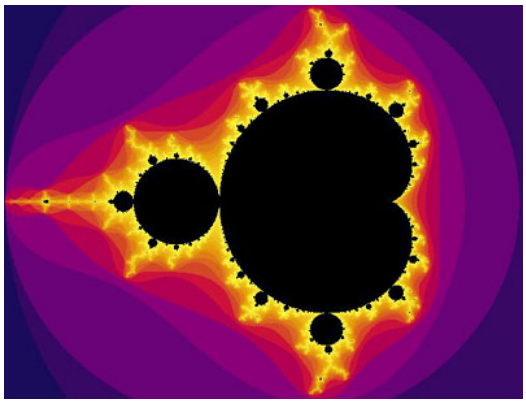

Figura 4.38: Conjunto de Mandelbrot

El bucle que aplica este criterio de pertenencia para cada punto es el que se muestra en la línea 11 del listado 4.26. La aplicación del mismo a cada punto da lugar a tareas de muy diverso peso, no pudiéndose determinar a priori cuántas iteraciones van a ser necesarias, pudiendo ser unas pocas o muchas para los puntos externos (se cumple la condición de parada de la línea 15), mientras que será necesario aplicar el máximo de iteraciones establecido (`MAXITER`, línea 11) para los puntos internos. Éste es un buen escenario para aplicar el paralelismo de cola de tareas.

```
numoutside = 0;
#pragma intel omp parallel taskq
for(i = 0; i < npoints; i++) {
#pragma intel omp task
#pragma llc task_master_data (&i, 1)
#pragma llc task_t_reduce_slave (&numoutside,int,1,LLC_SUM)
#pragma llc task_slave_set_data (&numoutside, 1, 0)
{
  z.creal = c[i].creal;
  z.cimag = c[i].cimag;
  for (j = 0; j < MAXITER; j++) {
    ztemp = (z.creal*z.creal)-(z.cimag*z.cimag)+c[i].creal;
    z.cimag = z.creal * z.cimag * 2 + c[i].cimag;
    z.creal = ztemp;
    if (z.creal * z.creal + z.cimag * z.cimag > THRESOLD) {
      numoutside++;
      break;
    }
  } /* for j */
} /* task */
} /* for i */
numinside = npoints - numoutside;

 area= 2.0 * 2.5 * 1.125 * numinside / npoints;
 error = area / sqrt(npoints);
```

Listado 4.26: Paralelización del algoritmo `Mandelbrot` implementado usando `taskq`

En cuanto a las directivas utilizadas en el listado 4.26, todas han sido ya explicadas, y sólo cabe mencionar que el único valor que necesita enviar el maestro a los esclavos es la iteración actual (directiva `task_master_data`, línea 5) y, a su vez, los esclavos devuelven si ese punto pertenece o no al conjunto mediante la actualización de la suma parcial de la variable `numoutside`, por lo que es necesario aplicar una reducción de tipo suma (directiva `task_t_reduce_slave`, línea 6). Por último, ya que los esclavos "recuerdan" el valor de esta variable `numoutside` de la última tarea, es necesario reestablecerlo a su valor original en cada nueva tarea (directiva `task_slave_set_data`, línea 7).

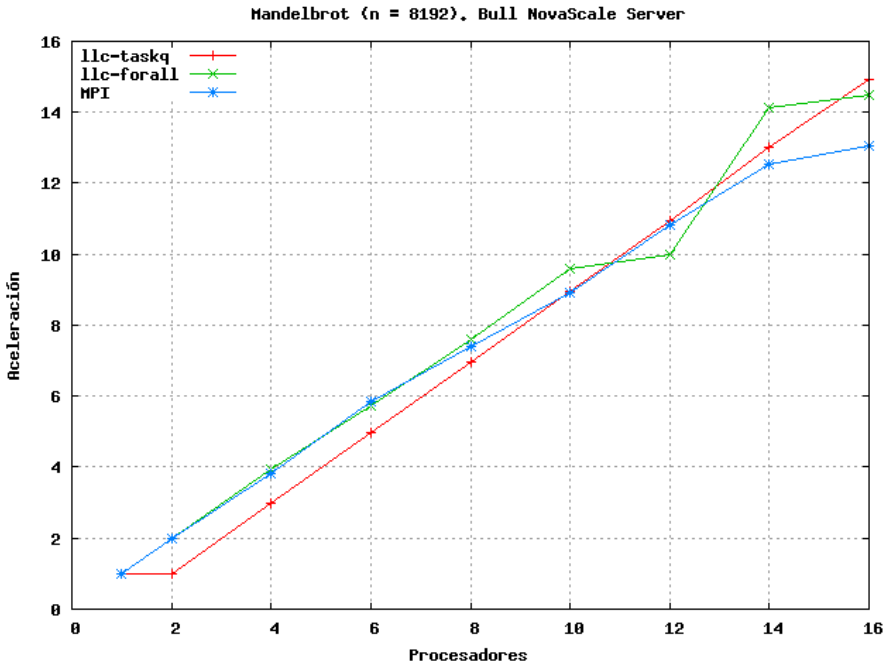

Figura 4.39: Aceleración para el código `Mandelbrot` en el `Bull NovaScale Server`

La figura 4.39 muestra los resultados obtenidos para este algoritmo en `Bull NovaScale Server` con 8192 puntos, comparándolo con la versión de bucles paralelos, tanto para la versión de `llc` ya presentada en el subapartado 4.2.7 como para el algoritmo *ad hoc* en `MPI`.

En esta figura 4.39 destaca cómo la curva de aceleración de la versión `taskq` ofrece un comportamiento más lineal y tiene una mayor pendiente que las correspondientes a otras versiones, por lo que para un número suficientemente grande de procesadores (12 en este caso), el rendimiento

que se obtiene es mayor, a la vez que es más estable, viéndose menos perjudicado por las variaciones que afectan a las otras versiones basadas en bucles paralelos.

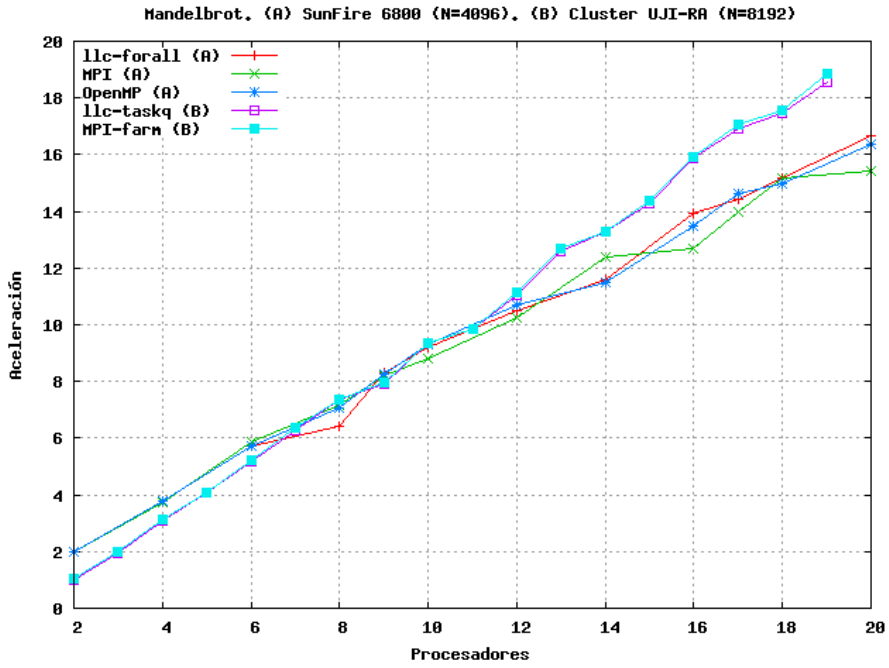

Figura 4.40: Aceleración para el código `Mandelbrot` en varias plataformas

La figura 4.40 muestra los resultados computacionales [60] obtenidos para el algoritmo `Mandelbrot` en varias plataformas y mediante diferentes aproximaciones. Tenemos dos grupos de resultados, por un lado los tomados en el `SunFire 6800` con 4096 puntos, marcados con `(A)`, y que corresponden versión tradicional del algoritmo con bucles paralelos tanto en `llc`, `OpenMP` y una versión *ad hoc* en `MPI`, habiéndose obtenido un rendimiento similar en estos tres casos. Por otro lado, los resultados señalados con `(B)` han sido tomados en el `Cluster UJI-RA` para 8192 puntos para la versión de colas de tareas de `llc` y una versión *ad hoc* en `MPI` con un enfoque de granja, ofreciendo ambas aproximaciones aproximadamente el mismo rendimiento, a pesar del diferente esfuerzo necesario para desarrollar el código en cada situación. Los tiempos a partir de los cuales se han obtenido las aceleraciones que muestra esta figura se recogen en la tabla 4.15.

	SunFire 6800			UJI-RA	
procs.	`llc for`	`MPI`	`OpenMP`	`llc taskq`	`MPI farm`
SEQ	6.28306			40.0721	
2	3.19216	3.16854	3.19261	40.18	38.8723
3				19.7406	19.4372
4	1.66339	1.67399	1.65961	12.9629	12.8529
5				9.74243	9.79614
6	1.09655	1.07399	1.10028	7.84281	7.73527
7				6.55769	6.44692
8	0.97613	0.87961	0.88817	5.57381	5.59383
9	0.75638	0.76744	0.75923	4.87251	4.85434
10	0.68411	0.71425	0.67634	4.30729	4.32651
11				3.89202	3.8838
12	0.59932	0.61395	0.58877	3.5402	3.51149
13				3.24108	3.22098
14	0.54306	0.50806	0.54768	3.00151	2.99415
15				2.79937	2.77599
16	0.45077	0.49621	0.46523	2.59548	2.59056
17	0.43526	0.44969	0.43033	2.44668	2.42192
18	0.41417	0.41418	0.41920	2.2983	2.28934
19				2.1872	2.1521
20	0.37753	0.40801	0.38394		

Tabla 4.15: Tiempos (en segundos) obtenidos para el código `Mandelbrot` con directiva `taskq`

4.5.3. Algoritmo `Microlensing` (con directiva `taskq`)

El algoritmo `Microlensing` ya estudiado en el apartado 4.2.8 tiene cierto parecido al algoritmo `Mandelbrot` estudiado en el anterior apartado 4.5.2, en el sentido que ambos tienen un bucle interno que, dependiendo de una condición, realizará un mayor o menor número de cálculos, lo cual da lugar a un desequilibrio en la carga de las tareas computadas. Por este motivo, hemos procedido a paralelizar el algoritmo `Microlensing` siguiendo un modelo de cola de tareas.

El listado 4.27 muestra el resultado de la paralelización del bucle externo mediante el constructo `taskq`. Hemos ocultado el contenido del bucle interno de la línea 19 por motivos de espacio, pero el lector puede consultarlo a partir de la línea 3 del listado 4.10 de la página 181.

Para esta paralelización, el maestro sólo necesita enviar al esclavo el índice que contiene la iteración actual (directiva `task_master_data`, línea 4), mientras que el esclavo devuelve como resultado el vector `s`, al que el maestro debe aplicar una operación de reducción de tipo suma (directiva `task_t_reduce_slave`, línea 6). Para evitar que los valores obtenidos en ejecuciones anteriores interfieran en el desarrollo de la tarea actual, el vector de resultados se restaura al valor inicial en la ejecución de cada tarea como indica la directiva `task_slave_set_data` de la línea 5.

La figura 4.41 presenta los resultados obtenidos para este algoritmo en `IBM RS-6000` [103] para un problema de las mismas características que el presentado en el apartado 4.2.8, con un número total de ecuaciones aproximado de 0.23 billones. Esta figura 4.41 cuenta con la versión de colas de tareas de `llc` (etiqueta `llc taskq`) y una versión *ad hoc* en `MPI` con una granja de tareas. Para este caso el rendimiento de `llc` es comparable con las versiones que paralelizan el bucle externo de la aplicación, mientras que la versión de `MPI` ofrece pobres resultados.

El mismo experimento se muestra en la figura 4.42 ejecutado esta vez en el `Cluster IAC`, con unos resultados prácticamente idénticos para las versiones que paralelizan el bucle externo y el enfoque de cola de tareas o granja, tanto para `llc` como para `MPI`. También destaca cómo el rendimiento de las paralelizaciones de los bucles internos adolecen de baja escalabilidad, viéndose ésta gravemente afectada por el mayor número de comunicaciones y el menor grano de las tareas. Los tiempos de cada una de las versiones de esta figura se recogen en la tabla 4.16.

```
1  #pragma llc parallel taskq
2    for ( i = 0; i < nrayx_i; ++i) {
3  #pragma llc task
4  #pragma llc task_master_data(&i, 1)
5  #pragma llc task_slave_set_data (s, size, 0.0)
6  #pragma llc task_t_reduce_slave (s, float, size, LLC_SUM)

8      {
9        xx = -lplsx + (float)i * dist;

11       for ( isum = 0; isum < nlen; ++isum ) {
12         x = xx - xpos[isum];
13         field[isum].sqrx = x * x;
14         field[isum].xrat = x * rat[isum];
15       }
16       kgx = kg1 * xx - xrfmin;

18       for ( j = 0; j < nrayy_i; ++j ) {
19         ...
20       }
21     } /* TASK */
22   }
```

Listado 4.27: Paralelización del algoritmo `Microlensing` implementado usando `taskq`

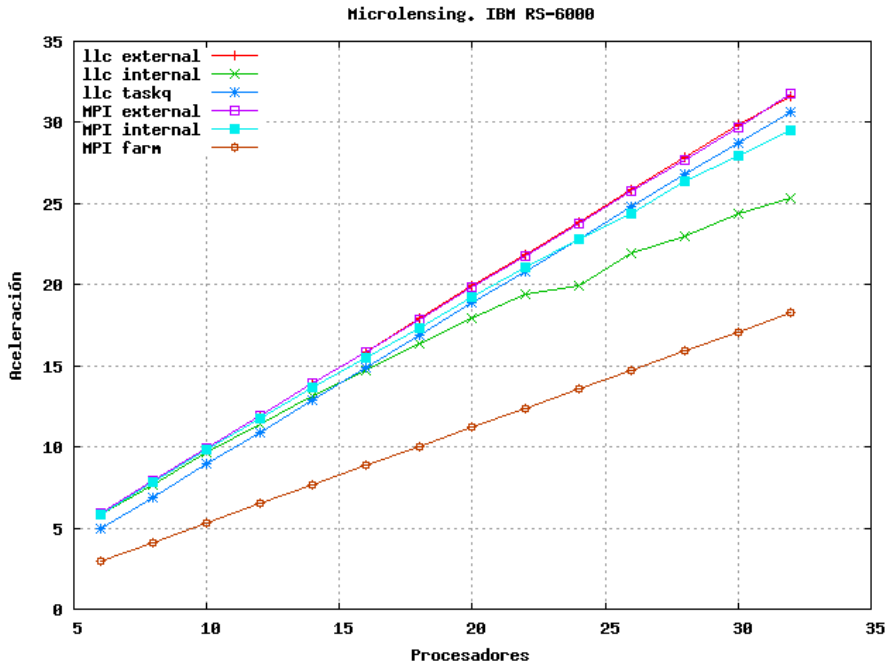

Figura 4.41: Aceleración para el código `microlensing` en `IBM RS-6000`

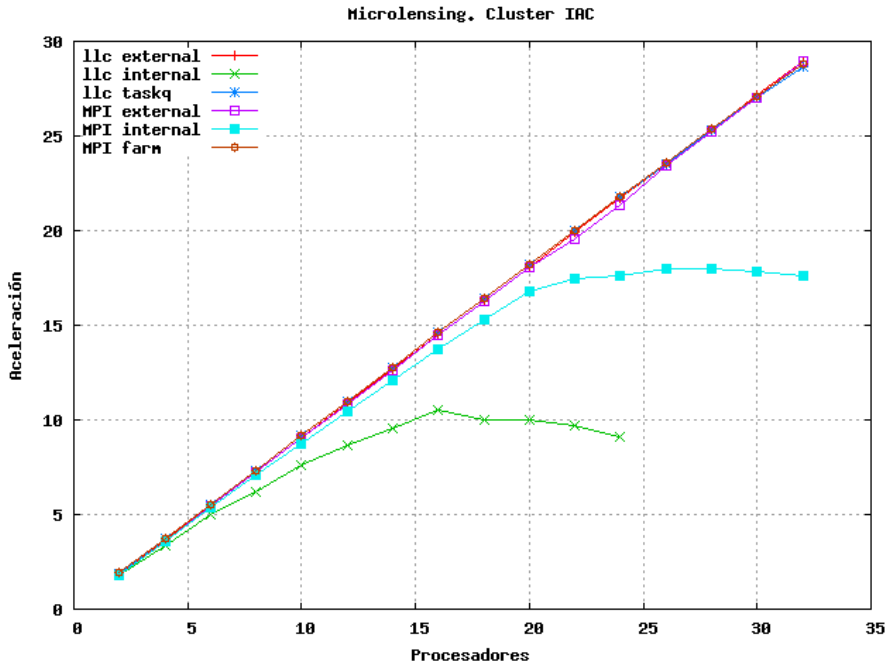

Figura 4.42: Aceleración para el código `microlensing` en `Cluster IAC`

	Cluster IAC					
procs.	`llc ext`	`MPI ext`	`llc int`	`MPI int`	`llc taskq`	`MPI farm`
SEQ	123786.52					
2	68317.7	68386.8	69316.8	68481.6	65162.5	65042.0
4	34165.3	34215.0	36561.6	34482.8	33308.7	33487.1
6	22776.4	22798.8	24938.5	23134.0	22378.4	22374.1
8	17080.1	17091.6	19920.1	17437.2	16852.5	16848.0
10	13668.9	13703.2	16234.4	14154.4	13513.0	13510.5
12	11392.5	11411.4	14350.5	11842.1	11279.4	11278.6
14	9766.2	9787.2	13008.3	10214.5	9679.2	9679.5
16	8541.4	8546.2	11796.3	9024.5	8479.2	8478.9
18	7598.7	7618.8	12353.5	8108.3	7544.1	7548.3
20	6840.3	6859.9	12411.1	7370.4	6795.5	6800.1
22	6215.2	6325.6	12750.2	7078.3	6189.7	6185.8
24	5697.4	5808.3	13629.6	7016.6	5684.7	5676.0
26	5264.7	5283.5		6891.1	5261.4	5247.2
28	4887.8	4908.2		6877.9	4896.9	4883.5
30	4562.6	4583.4		6937.9	4583.2	4564.3
32	4271.1	4275.4		7033.2	4318.0	4292.2

Tabla 4.16: Tiempos (en segundos) obtenidos para el código `microlensing`

4.5.4. Algoritmo `syrk`

La operación `syrk` (*symmetric rank-k update*) es una de las rutinas más usadas de BLAS (*Basic Linear Algebra Subprograms*) [96, 48]. Un ejemplo de su aplicación es la formación de las ecuaciones normales en problemas lineales de mínimos cuadrados y la solución de sistemas lineales simétricos definidos positivos usando la factorización de Cholesky [77]. La operación computa la parte triangular inferior (o superior) del resultado del producto de matrices $C := \beta C + \alpha A A^T$, donde C es una matriz simétrica $m \times m$, A es una matriz $m \times k$ y α y β son escalares.

Para la paralelización de la operación `syrk` hemos tomado como punto de partida un código del proyecto `FLAME` (**F**ormal **L**inear **A**lgebra **M**ethod **E**nvironment) [12]. Este proyecto, además, nos ofrece la posibilidad de paralelización de varias operaciones de álgebra lineal densa [152]. Una de las características más destacables de `FLAME` es su capacidad para enmascarar indexaciones complejas en cálculos de álgebra lineal, lo cual puede ser interesante para reducir la complejidad de las operaciones, pero imposibilita la aplicación de bucles paralelos tipo `for` de `OpenMP`.

Por otro lado, el bucle principal de la operación depende de una condición de parada determinada por el tamaño de las submatrices, por lo que se utiliza un bucle `while` que evalúa esta condición. Estos motivos determinaron que la paralelización de esta operación la hayamos llevado a cabo mediante el constructo `taskq`.

El listado 4.29 muestra la paralelización final obtenida. A primera vista, ésta puede parecer compleja, sin embargo, se trata de un proceso de paralelización directo, si bien es necesario aplicar varias directivas debido no a la dificultad del algoritmo, sino a la complejidad de las estructuras que `FLAME` utiliza para representar los datos, donde cada una de las variables que contienen a las matrices son consideradas como objetos.

El listado 4.28 muestra la definición de estas estructuras de tipo `FLA_Obj` y ha sido incluido para ayudar a la comprensión del lector de las directivas usadas. Entre los campos, caben destacar el puntero al objeto que contiene los datos (`base`) y el propio puntero a los datos (`buffer`), así como los valores de los índices del tamaño de las matrices (`m` y `n`), los desplazamientos (`offn` y `offm`) y la dimensión (`ldim`). El uso de estas estructuras tiene como efecto un aumento en el número y complejidad de las directivas de `llc` con respecto a otras paralelizaciones estudiadas.

Las operaciones que se desarrollan en el bucle corresponden con $C_{10} =$

```
1  struct FLA_Obj_struct{
2    int datatype;
3    int m;
4    int n;
5    int ldim;
6    int offset;
7    void *buffer;
8    int fp;
9  };

11 typedef struct FLA_Obj_struct FLA_Base_obj;

13 struct FLA_Obj_view{
14   int offm, offn;
15   int m, n;
16   FLA_Base_obj *base;
17   void* buffer;
18 };

20 typedef struct FLA_Obj_view FLA_Obj;
```

Listado 4.28: Definición de las estructuras `FLA_Obj`

$C_{10}+A_1\times {A_0}^T$ y $C_{11} = C_{11}+A_1\times {A_1}^T$, siendo necesario que el maestro envíe a los esclavos los valores del lado derecho, en concreto las submatrices afectadas `C10`, `C11`, `A0` y `A1`. Sin embargo, al estar los datos replicados en todos los procesadores, todos ellos contienen los valores de las matrices, por lo que sólo es necesario que el maestro comunique a los esclavos los correspondientes valores de los índices (`m` y `n`) y los desplazamientos (`offm` y `offn`) según se necesite, siendo el número de datos total a comunicar tan sólo de 7 enteros.

Estas comunicaciones se especifican mediante las directivas `task_master_data` de las líneas 7 a 11 del listado 4.29. Por su parte, los esclavos han de devolver los resultados de la parte izquierda de la expresión, es decir, las submatrices `C10` y `C11`, lo que se realiza a través de las dos directivas `task_slave_rnc_data` de las líneas 20 a 27, que comunican los datos de las submatrices con accesos no contiguos, pero que siguen un patrón regular.

Por último, cada tarea modifica algunos de los campos de los objetos durante la ejecución de cada tarea (como, por ejemplo, los punteros bases y los valores de los índices y desplazamientos) por lo que se hace necesario restaurar estos campos a los valores iniciales al ejecutar cada nueva tarea. De esto se encargan las directivas `task_slave_set_data` de las líneas 12 a 19, que copian los valores correctos de los objetos iniciales `A` y `C` a los correspondientes objetos modificados `A0`, `A1`, `C10` y `C11`, realizándose esta operación localmente sin que se requiera ninguna comunicación.

Una de las razones por la que hemos implementado este algoritmo en `llc` es que se disponía de la versión paralela en `OpenMP` utilizando las directivas de `Intel-KAI`, así como un compilador `Intel` que ofrecía soporte a estas directivas. Esto nos permitía comparar los resultados obtenidos con `llc` con los de `OpenMP`. También se ha implementado la paralelización de una segunda variante conceptualmente similar, que sólo se diferencia en las las operaciones afectadas, siendo en esta segunda variante $C_{21} = C_{21} + A_2 \times {A_1}^T$ y $C_{11} = C_{11} + A_1 \times {A_1}^T$.

La figura 4.43 muestra los resultados obtenidos en el `Bull NovaScale Server` para un problema con $m = 10000, k = 7000$ y un tamaño de bloque de 384, para dos de las variantes del algoritmo. Como se puede observar, los resultados son similares tanto para `llc` como para `OpenMP`. Sin embargo, la versión de `OpenMP` se ve afectada en mayor medida por problemas de ancho de banda, lo que disminuye su rendimiento. De este modo, a partir de un número de procesadores (concretamente a partir de 7), la versión de cola de tareas de `llc` supera en aceleración a la de `OpenMP`.

```
#pragma intel omp parallel taskq
{
  while (FLA_Obj_length(AT) < FLA_Obj_length(A)){
  ...

#pragma intel omp task
#pragma llc task_master_data(&A0.m,1,&A1.offm,1,&A1.m,1)
#pragma llc task_master_data(&C11.offm,1,&C11.offn,1, \
                             &C11.m,1,&C11.n,1)
#pragma llc task_master_data(&C10.offm,1,&C10.offn,1, \
                             &C10.m,1,&C10.n,1)
#pragma llc task_slave_set_data(&A1.base,1,A.base, \
                                &A0.base,1,A.base)
#pragma llc task_slave_set_data(&C11.base,1,C.base, \
                                &C10.base,1,C.base)
#pragma llc task_slave_set_data(&A0.offm,1,A.offm, \
                      &A0.offn,1, A.offn,&A0.n,1,A.n)
#pragma llc task_slave_set_data(&A1.offn,1,A.offn, \
                                &A1.n,1,A.n)
#pragma llc task_slave_rnc_data ((C10.base->buffer+ \
        ((C10.offn*C10.base->ldim+C10.offm) * \
        sizeof(double))), (C10.m * sizeof(double)), \
        ((C10.base->ldim - C10.m) * sizeof(double)), C10.n)
#pragma llc task_slave_rnc_data ((C11.base->buffer+ \
   ((C11.offn*C11.base->ldim+C11.offm)*sizeof(double))), \
    (C11.m * sizeof(double)), ((C11.base->ldim - C11.m) * \
     sizeof(double)), C11.n)

    {
      /* C10 := C10 + A1 * A0' */
      FLA_Gemm(FLA_NO_TRANSPOSE, FLA_TRANSPOSE,
               alpha, A1, A0, beta, C10, nb_alg);
      /* C11 := C11 + A1 * A1' */
      FLA_Syrk(FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE,
               alpha, A1, beta, C11, nb_alg);
    }
  }
}
```

Listado 4.29: Código `FLAME` para la operación `syrk` paralelizada usando `llc`

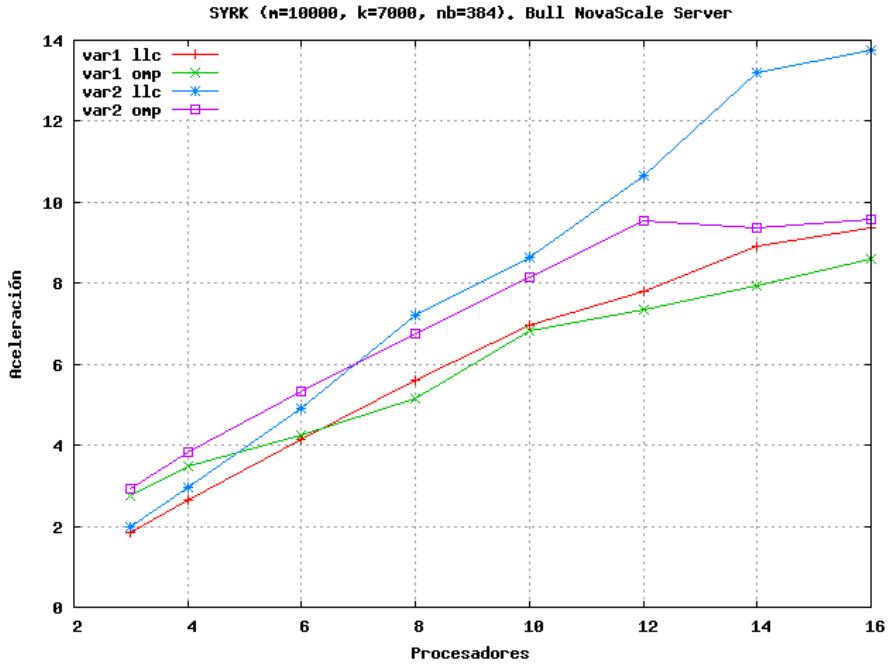

Figura 4.43: Aceleración para el código `syrk` en el `Bull NovaScale Server`

La figura 4.44, obtenida en `SGI Altix 250` [51, 53] para un problema con $m = 6000, k = 3000$ y un tamaño de bloque de 96, corrobora los resultados mostrados en la anterior figura 4.43. Las curvas de aceleración de la figura 4.44 siguen un comportamiento similar al caso anterior, siendo válidas las observaciones realizadas, si bien en este caso `llc` supera la aceleración de `OpenMP` para un valor más alto de procesadores (9 para la primera variante y 13 para la segunda). La tabla 4.17 recoge los tiempos a partir de los cuales han sido calculadas las curvas de aceleración de esta figura.

Por otro lado, ambas figuras 4.43 y 4.44 muestran cómo la segunda variante presenta un mejor rendimiento que la primera. Esto se debe a que la operación paralelizada en la segunda variante genera un número mayor de tareas con un mejor grano que da lugar a un mayor rendimiento paralelo [51]. La diferencia de rendimiento entre ambas variantes queda patente en la figura 4.45 obtenida en el `Cluster UJI-CAT` con $m = 5000, k = 3000$, tamaño de bloque de 96 y 3 repeticiones, escogiéndose la media de los tiempos para calcular el rendimiento.

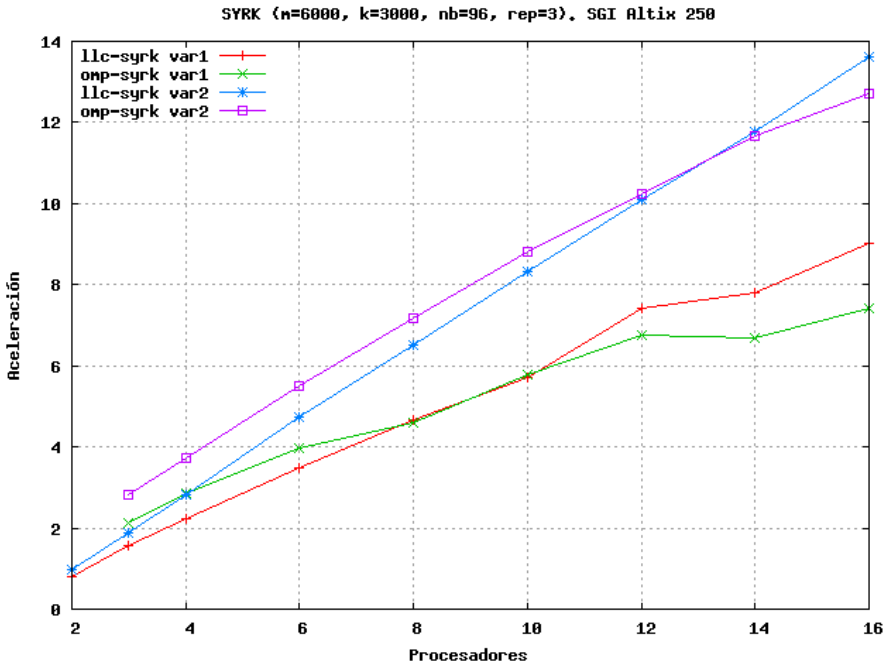

Figura 4.44: Aceleración para el código `syrk` en la `SGI Altix 250`

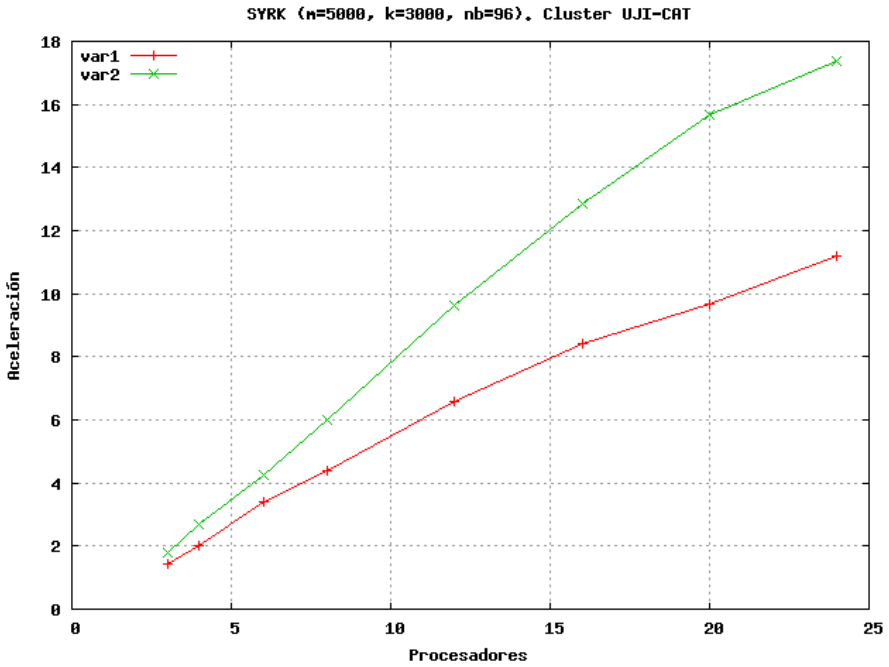

Figura 4.45: Aceleración para el código `syrk` en el `Cluster UJI-CAT`

	SGI Altix 250			
procs.	omp-var1	omp-var1	llc-var2	omp-var2
SEQ	19.00			
2	24.0057		19.7649	
3	12.0147	8.9384	10.0298	6.7211
4	8.54865	6.66211	6.7268	5.10677
6	5.46592	4.77383	4.04035	3.44234
8	4.05341	4.12401	2.91102	2.6508
10	3.33226	3.28062	2.28011	2.15013
12	2.56287	2.80562	1.88201	1.85261
14	2.42966	2.83464	1.61174	1.62577
16	2.10784	2.56164	1.39658	1.49218

Tabla 4.17: Tiempos (en segundos) obtenidos para el código `syrk`

4.5.5. Algoritmo `gemv`

La operación `GEMV` (*GEneral Matrix-Vector*) [48] es una de las operaciones `FLAME` que computa el producto matriz por vector según la expresión $y = \beta y + \alpha Ax$, donde x e y son vectores, A una matriz y α y β escalares.

El listado 4.30 muestra la paralelización de esta operación, realizada de un modo similar a la presentada para la operación `syrk` en el apartado 4.5.4 (página 234). En este caso, también se utilizan los objetos `FLA_Obj` cuya definición se muestra en el listado 4.28 (página 235). No obstante, al tratarse de una operación en la que mayoritariamente intervienen vectores, en este caso el resultado de la paralelización es más simple que el de la operación `syrk`.

De esta forma, el maestro sólo debe comunicar a los esclavos el índice y desplazamiento de las variables `y1` y `A1`, mediante las directivas `task_master_data` de las líneas 6 a 8 del listado 4.30, y los esclavos enviarán al maestro los resultados parciales a través de las regiones de memoria del vector `y1` que han sido modificadas. Como, en este caso, estas regiones son contiguas, se utiliza la directiva `task_slave_data` de las líneas 16 y 17. La inicialización de los punteros bases, índices y desplazamientos para las variables `y1` y `A1` antes de ejecutar una nueva tarea se especifica mediante las directivas `task_slave_set_data` de las líneas 9 a la 14.

La figura 4.46 presenta los resultados computacionales obtenidos para el algoritmo `gemv` en `SGI Altix 250` con un tamaño de problema de $m = 17000$

```
#pragma intel omp parallel taskq
{
  while (FLA_Obj_length(AT) < FLA_Obj_length(A)) {
  ...
#pragma intel omp task
#pragma llc task_master_data (&A1.m, 1, &A1.offm, 1)
#pragma llc task_master_data (&y1.m, 1, &y1.offm, 1)

#pragma llc task_slave_set_data (&A1.base, 1, A.base, \
                                 &y1.base, 1, y.base)
#pragma llc task_slave_set_data (&A1.offn, 1, A.offn, \
                                 &A1.n, 1, A.n)
#pragma llc task_slave_set_data (&y1.offn, 1, y.offn, \
                                 &y1.n, 1, y.n)

#pragma llc task_slave_data ((y1.base->buffer + \
          (y1.offm*sizeof(double))), (y1.m * sizeof(double))
    )
    {
      /* y1 = y1 + A1 * x */
      FLA_Gemv (FLA_NO_TRANSPOSE, alpha, A1, x, beta, y1);
    }
    ...
  }
}
```

Listado 4.30: Paralelización del algoritmo `gemv` implementado usando `taskq`

(matriz $m \times$m) y $k = 14000$ (vector de k elementos). El tamaño de bloque (nb) elegido es de $nb = 384$ y el algoritmo se repitió tres veces, tomando los tiempos medios (que se muestran en la tabla 4.18), para reducir la posibilidad de aparición de fluctuaciones. La curva de aceleración tiene una tendencia marcadamente lineal, si bien existe un valor discordante al ejecutar con 14 procesadores. Esto puede deberse a la desasignación de algún procesador durante una de las ejecuciones, lo cual pudo producir una disonancia más realzada, pero que ha quedado parcialmente mitigada al realizar tres ejecuciones.

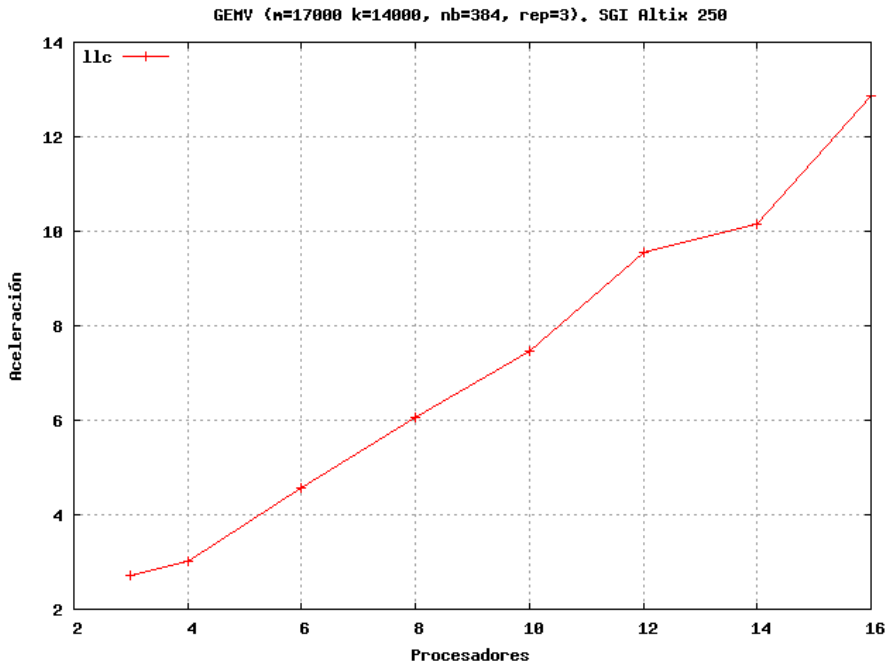

Figura 4.46: Aceleración para el código `gemv` en la `SGI Altix 250`

procs.	`SGI Altix 250`
SEQ	0.742420
3	0.278635
4	0.246805
6	0.162357
8	0.123464
10	0.100042
12	0.077835
14	0.073779
16	0.054780

Tabla 4.18: Tiempos (en segundos) obtenidos para el código `gemv`

4.6. Otros trabajos: OmpSCR

La necesidad de validar nuestra implementación mediante la toma de resultados computacionales nos llevó a una búsqueda de códigos OpenMP que otros grupos de investigación con aproximaciones similares a la nuestra estuvieran desarrollando y evaluando. Nuestro objetivo era el de ejecutar estos códigos con las mismas condiciones y parámetros con el fin de obtener una estimación lo más fiable posible del rendimiento de nuestra aproximación con respecto a otras similares.

Sin embargo, encontramos que, salvo ciertos casos obvios con aplicaciones muy conocidas, era extraña la situación en la que los códigos fuente usados en las medidas de rendimiento estuvieran a libre disposición, y en más rara ocasión se tenía acceso a todos los detalles de la ejecución (valores de los parámetros, etc.) con el fin de poder reproducir el experimento.

Por este motivo, llegamos a la conclusión que sería de utilidad disponer de una infraestructura abierta, no propietaria y libremente accesible donde todos los usuarios que lo deseasen pudieran obtener los códigos ejecutados en los experimentos y conocer los resultados computacionales de estas aplicaciones y las circunstancias en las que fueron obtenidos.

Para dar respuesta a esta necesidad, en paralelo con el proceso de investigación de esta tesis desarrollamos un repositorio de código fuente de aplicaciones paralelas en OpenMP, que denominamos OmpSCR (OpenMP **S**ource **C**ode **R**epository) [57, 59]. Este repositorio se implementó a través de una plataforma web que permitía que los usuarios pudieran cargar y descargar sus códigos.

Sin embargo, no sólo pretendíamos que esta web sirviera meramente como un lugar de intercambio de códigos, sino que fuera un espacio activo y un punto de encuentro para toda la comunidad científica interesada. Para ello se habilitaron foros de discusión, se permitió que los usuarios pudieran subir sus gráficas, ficheros de resultados, artículos, publicaciones, documentación, etc. con el fin de poder distribuirlos y compartirlos.

En definitiva, se intentó crear un marco de trabajo que fuera útil para cualquier persona interesada en programar en OpenMP, independientemente de su nivel, desde el principiante que podría descargarse sus primeros códigos desde el sitio web o formular preguntas, hasta los usuarios avanzados que podrían cargar sus resultados y discutirlos, proponer nuevas paralelizaciones, resolver dudas, etc.

El repositorio OmpSCR ha sido presentado en varios congresos [57, 59],

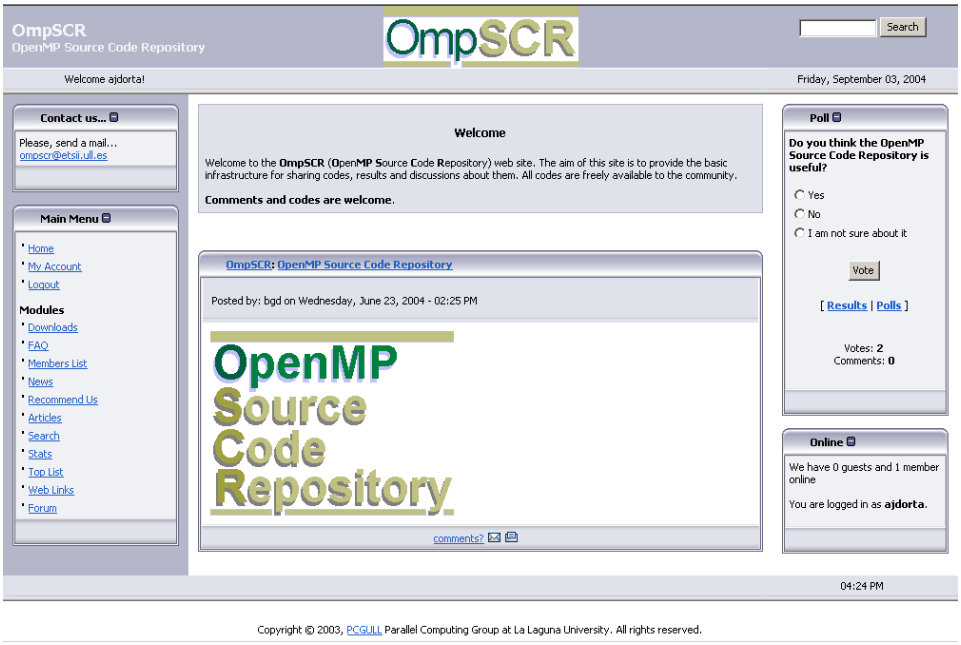

Figura 4.47: Portada de OmpSCR

contando con una buena aceptación entre la comunidad e incluso siendo propuesto como una parte integrante del portal de la comunidad de OpenMP [37].

La figura 4.47 muestra la portada de esta web en su origen. Sin embargo, una de las sugerencias que se recibieron durante la presentación del repositorio es que éste debería estar enmarcado dentro de algún proyecto de mayor recorrido, así que evaluamos distintas alternativas y finalmente el repositorio OmpSCR fue integrado como un proyecto de SourceForge [121], sugerencia aportada por el profesor Dieter An Mey, de donde es posible obtener la última versión 2.0 de los códigos.

Además del trabajo directamente relacionado con llc, durante la fase de desarrollo de esta tesis también se abordaron otros temas vinculados, como por ejemplo en el campo de la enseñanza [54].

Capítulo 5

Conclusiones y Trabajos Futuros

5.1. Conclusiones

Hemos comenzado esta memoria señalando la necesidad de facilitar la programación paralela a usuarios que requieren su utilización, pero carecen de los conocimientos necesarios para poder hacer uso de ella, en muchas ocasiones porque provienen de contextos científicos diversos poco o nada relacionados con la computación.

Por otro lado, en la actualidad las tecnologías de CAP están dejando de tener su ámbito casi exclusivo en el mundo científico para ir paulatinamente asentándose también en la computación doméstica y de propósito general. Así lo demuestran las emergentes investigaciones en el campo de la tecnología *multicore* [6], que mediante la aplicación del paralelismo buscan extraer el máximo rendimiento a los microprocesadores que contienen varios núcleos, recientemente convertidos en el nuevo estándar de fabricación en la industria de procesadores.

Si deseamos que el paralelismo dé servicio a la gran diversidad de usuarios que potencialmente lo demandan, es nuestro deber como científicos de la computación el poner a su disposición lenguajes y modelos de programación accesibles y eficientes.

Desde nuestro punto de vista, las aproximaciones que se acerquen a este objetivo y que tengan posibilidades reales de convertirse en estándares de programación paralela en los próximos años deben cumplir, además de con las características obvias de eficiencia y rendimiento, al menos con dos requisitos

más:

1. *Portabilidad*. Los lenguajes deben unificar y garantizar una *independencia* de la arquitectura final en la que se ejecutará la aplicación.

2. *Simplicidad* de uso. Hemos de tener en cuenta que la demanda real de paralelismo proviene predominantemente de una comunidad que no tiene, ni debe tener, conocimientos avanzados en este tópico. Esta comunidad ya ha realizado un esfuerzo considerable en adquirir la preparación necesaria para desarrollar programas secuenciales y, por este motivo, programar en paralelo debería suponer una suave transición para ellos. El objetivo a perseguir es el de una programación paralela simple, y en lo posible similar a la secuencial.

Consideramos que en la actualidad no existen muchas alternativas que satisfagan estas condiciones, si bien recientemente se ha desarrollado un gran interés por lograr este objetivo y diversas investigaciones avanzan en este sentido. A lo largo del primer capítulo de esta memoria realizamos un recorrido por los diferentes lenguajes que han marcado la programación paralela en el pasado reciente, en el presente y en un futuro próximo. La mayoría de lenguajes que actualmente se encuentran en fase de investigación dan respuesta en mayor o menor medida al primero de los requisitos, si bien no está tan claro el logro del segundo.

De hecho, un aspecto negativo que hemos constatado al realizar la revisión de estos lenguajes es que, a pesar de que el objetivo parece estar razonablemente definido y hay un gran esfuerzo para que se logre, como el programa HPCS de DARPA, existe una gran dispersión en cuanto a las investigaciones que se están desarrollando, posiblemente producida por los enormes intereses económicos subyacentes.

No parece existir una coordinación entre los distintos proyectos, mientras que empresas y universidades han creado grupos que desarrollan sus propios lenguajes que, en la mayoría de casos, sólo dan una respuesta parcial a los problemas planteados, en lugar de aunar esfuerzos en una línea compartida de trabajo.

Ni siquiera se atisba un principio de acuerdo en la orientación que se va a seguir, y el conjunto de lenguajes que actualmente se desarrollan recoge los más diversos enfoques, siendo algunos de ellos lenguajes totalmente nuevos, otros se plantean como modificaciones o extensiones de lenguajes secuenciales

existentes, algunos dan soporte a paralelismo de carácter general, otros tan sólo están orientados a campos específicos, etc.

Esta situación no favorece a la comunidad que en la actualidad desee adoptar las ventajas a la programación paralela. Lo primero que encuentra un neófito en la materia es la dificultad de la elección del lenguaje en el que debe implementar sus aplicaciones, ya que existe un amplio conjunto, sin que haya el menor atisbo sobre qué proyectos van a tener un peso específico en un futuro próximo y cuáles desaparecerán en breve, arrastrando con ellos las aplicaciones que hayan sido desarrolladas según su modelo.

Aunque unos lenguajes suenen con más fuerza que otros, su futuro aún es incierto y está por decidir cuál será el que ocupe el estándar de programación paralela en los próximos años, siendo probablemente la "selección natural" de la propia comunidad la que lleve a cabo esta elección.

En este sentido, `llc` apuesta enérgicamente por los dos puntos que hemos destacado, como ha sido estudiado en profundidad en el segundo capítulo, donde se defiende y demuestra la viabilidad de la idea de diseñar un lenguaje paralelo independiente de la arquitectura, que permita programar de forma similar a la secuencial. Por este motivo, presentamos nuestra propuesta como una aportación al panorama presente de desarrollo de aplicaciones paralelas, proponiéndola como una solución factible a las dificultades actuales de esta tecnología.

El modelo OTOSP le confiere a nuestro lenguaje una gran simplicidad, lo que se traduce en un estilo de programación prácticamente similar al secuencial, siendo tan sólo necesario marcar aquellas regiones susceptibles de ser paralelizadas. El posterior proceso de generación del código paralelo final se lleva a cabo de forma transparente al usuario y con una mínima intervención del mismo. El resultado es un código paralelo `MPI` que es tan portable como lo es esta librería. Estas características son las principales cualidades de nuestra idea y representan una ventaja con respecto a algunas de las aproximaciones estudiadas en el primer capítulo.

Otro de los resultados de nuestro trabajo ha sido el desarrollo de un compilador que da soporte a nuestro lenguaje: `llCoMP`. Este compilador se encuentra implementado y disponible para prácticamente todas las plataformas que cuenten con un compilador de `C` y con herramientas básicas, como generadores de analizadores léxicos y sintácticos. `llCoMP` genera código paralelo eficiente y portable a partir de un código secuencial u `OpenMP` anotado con directivas de `llc`, para los diversos paradigmas de programación paralela que han sido estudiados en el tercer capítulo.

El hecho de disponer de un compilador implementado nos posibilita presentar nuestro trabajo avalado por los resultados computacionales que se han presentado en esta memoria. Hemos realizado un recorrido por algoritmos de diversas características, estudiados con diferentes aproximaciones y paradigmas de paralelización, bajo diferentes condiciones y parámetros y para un amplio conjunto de sistemas paralelos que abarcan una buena representación del panorama actual en las distintas arquitecturas.

La implementación de un compilador y la obtención de resultados computacionales son consideradas como ventajas porque le otorgan a nuestro trabajo un aval del que un gran porcentaje de las alternativas estudiadas no dispone en la actualidad. Consideramos que entre los lenguajes que hemos analizado existen aproximaciones interesantes, pero lamentablemente el conjunto de ellas que a día de hoy están disponibles y accesibles es aún reducido. La mayoría de los lenguajes estudiados se encuentran en un estado de discusión técnica o bien su implementación aún no es lo suficientemente madura o está reducida a determinados ámbitos, siendo muy pocos los lenguajes que justifiquen sus características mediante un análisis de resultados computacionales de una cierta magnitud.

Por último, también queremos reflejar nuestro esfuerzo para contribuir a la mejora de la infraestructura necesaria para la expansión de la CAP. En esta línea hemos desarrollado `OmpSCR`, un proyecto consistente en un repositorio de códigos fuente de `OpenMP` con el que esperamos haber aportado a la comunidad de usuarios un lugar común y punto de encuentro sobre el estudio de este tópico.

5.2. Trabajos Futuros

La investigación no siempre sigue un camino perfectamente trazado y es difícil establecer dónde se encuentra el principio y dónde el final. Por este motivo, ha sido necesario establecer unas cotas en contenido y tiempo para este trabajo.

Entre los tópicos que consideramos que podrían ser líneas naturales de continuidad de este trabajo, destacan:

- Implementar en nuestro compilador varias cláusulas de `OpenMP` que, si bien no tienen demasiada relevancia en nuestro modelo, sí aumentarían la compatibilidad. Entre estas cláusulas cabe mencionar: `if-parallel`, `num_threads`, `depth-level`, ...

- Estudiar e incorporar otros constructos de explotación de paralelismo.
- Ampliar el conjunto de algoritmos implementados con `llc`, haciendo especial énfasis en aquellos códigos de interés científico real.
- Realizar más experimentos computacionales y medidas de rendimiento.
- Llevar a cabo una vigilancia de otras aproximaciones similares a la nuestra y comparar nuestros resultados con ellas.
- Realizar análisis de rendimiento y ajuste de los códigos generados por nuestro compilador con herramientas como `Paraver` [125] y `Dimemás` [46].
- Estudiar el nuevo estándar de `OpenMP` 3.0. Considerar la incorporación e implementación de las nuevas directivas en nuestro lenguaje. La aprobación de este estándar (mayo de 2008) se produjo cuando esta memoria ya estaba avanzada.
- Incorporar en el compilador una fase de análisis semántico y comprobación de tipos.
- Estudiar la posibilidad de una migración del lenguaje y compilador a `C++` y a un modelo orientado a objetos.

Apéndice A

Recursos computacionales

A.1. Introducción

Para el desarrollo de nuestro trabajo ha sido fundamental el uso de distintos sistemas en los que hemos obtenido los resultados computacionales mostrados en el capítulo 4. Este apéndice presenta un resumen de las principales características de los equipos que hemos usado. Citaremos previamente los distintos programas y proyectos que nos han permitido el acceso a estos sistemas.

El acceso a las máquinas del Centro de Supercomputación de Edimburgo (EPCC) [64] fue posible gracias a la estancia que realicé como doctorando en sus instalaciones durante los meses de septiembre y octubre de 2005, como parte de la formación como investigador y a través del programa *HPC-Europa* [84].

El uso de los equipos del Grupo de Computación Científica Paralela (PSCOM) [129] de la Universidad Jaime I de Castellón [147] ha sido posible gracias a la participación de nuestro grupo en diversos proyectos [3, 133], colaboraciones y una estancia con dicho grupo.

Otras colaboraciones y contacto con científicos de otros organismos nos han permitido también el acceso a equipos del Instituto de Astrofísica de Canarias (IAC) [87], del Centro Europeo de Paralelismo de Barcelona (CEPBA) [23] de la Universidad Politécnica de Cataluña (UPC) [150] y del Centro de Investigaciones Energéticas, Medioambientales y Tecnológicas (CIEMAT) [32].

Por último, debemos mencionar los propios equipos de la Universidad de

La Laguna [148], tanto los pertenecientes al Grupo de Paralelismo de la ULL (PCG-ULL) [127], como al Servicio de Apoyo Informático a la Investigación (SAII) [135].

A.2. Sistemas utilizados

Presentaremos aquí los sistemas con los que hemos trabajado y que nos han permitido obtener los resultados computacionales que avalan nuestra implementación.

Nuestro objetivo es el de demostrar la generalidad de nuestros resultados, por eso hemos intentado obtenerlos en el mayor conjunto de equipos posible, abarcando diferentes arquitecturas y plataformas. Entre la colección de sistemas con los que hemos trabajado podemos citar máquinas de memoria compartida, de memoria distribuida y equipos híbridos o mixtos.

Debemos mencionar que las configuraciones de los equipos que se indican en este apartado son únicamente orientativas, puesto que generalmente las máquinas son actualizadas cada cierto tiempo y sus características varían a lo largo de su vida útil, estando algunos de los equipos utilizados descatalogados o fuera de uso en la actualidad. En lo posible, hemos tratado de presentar en este apéndice las configuraciones más aproximadas a las que fueron usadas durante la toma de resultados.

A.2.1. `Bull NovaScale Server`

El sistema `Bull NovaScale Server` dispone de 64 procesadores Intel Itanium 2 a 1.5GHz, lo que le confiere una potencia total teórica por encima de los 300 GFlops. Su nombre lógico es `tarja` y pertenece al SAII de la Universidad de La Laguna. La máquina está dividida entre nodos de cómputo (11 servidores con 56 procesadores en diferentes particiones configurables) y un nodo de servicio (gestión y E/S), con dos servidores de 4 procesadores cada uno).

La capacidad bruta de almacenamiento es de unos 10 TB y la red de comunicaciones es una `Quadrics QsNetII` entre los nodos de cómputo y `Ethernet` para la gestión y administración. El sistema operativo es BAS3 (Bull Advanced Server) V1.5, paquete de software HPC de Bull compuesto por Linux Kernel 2.6.7, distribución Linux BAS3 (compatible con Red-Hat AS 3) y Bull extensions pack para HPC Linux 5.2, entre otros.

En este equipo se obtuvieron los resultados presentados en las figuras 4.19 (p. 185), 4.37 (p. 224) 4.39 (p. 227) y 4.43 (p. 238).

A.2.2. Cluster EPCC

El Cluster de nombre lógico `ness`, localizado en el EPCC, está formado por procesadores 2.6 GHz AMD Opteron (AMD64e) con 2 GB de memoria. El sistema está organizado en un *front-end* compuesto por 2 procesadores con fines de gestión, edición y compilación de los programas, y un *back-end* de 32 procesadores destinados a la ejecución. Estos 32 procesadores se agrupan en dos divisiones SMP (procesamiento simétrico - memoria compartida), conteniendo cada una 16 procesadores (8 nodos duales). El sistema operativo es Linux ("`Scientific Linux`").

En este sistema se tomaron los resultados que se recogen en las figuras 4.4 (p. 158) y 4.8 (p. 165).

A.2.3. Cluster IAC

Cluster perteneciente al Instituto de Astrofísica de Canarias (IAC), de nombre lógico `chimera`. Está formado por un nodo maestro y 32 nodos esclavos que se clasifican en dos grupos: por un lado, existen 16 nodos compuestos por máquinas de 64 bits con un total de 64 GB de RAM y 32 procesadores Intel Xeon 3.20GHz EM64, y, por otro lado, 16 nodos formados por máquinas i686, con un total de 32 GB de RAM y 32 procesadores Intel Xeon a 2.80GHz. El espacio total de disco es aproximadamente de 4 TB y el sistema cuenta con dos redes independientes `Gigabit Ethernet`.

Este cluster fue utilizado para obtener los resultados de la figura 4.42 (p. 232).

A.2.4. Cluster UJI-CAT

Cluster de nombre lógico `cat` que pertenece al PSCOM de la UJI, compuesto por un servidor y 9 nodos de trabajo, todos ellos tetraprocesadores, basados en procesadores Intel Itanium2 (arquitectura IA64) a 1.5 GHz. Cada nodo tetraprocesador posee memoria RAM compartida de 4 GB (8 GB en el caso del servidor) y la comunicación entre nodos puede desarrollarse mediante una red `InfiniBand` a 10 Gbit/s o una

red `Gigabit Ethernet`. El sistema operativo es Linux tanto en el servidor como en los nodos de trabajo, con los compiladores `gcc` e `icc`.

La figura 4.45 (p. 239) presenta los resultados computacionales obtenidos en este cluster.

A.2.5. `Cluster UJI-RA`

Este sistema, de nombre lógico `ra` y perteneciente al PSCOM de la UJI, está compuesto por un servidor y 34 nodos de trabajo, todos ellos biprocesadores de 32 bits Intel Xeon a 2.4 GHz y memoria de 1 GB DDR RAM por nodo biprocesador, compartida entre los procesadores del nodo. Las comunicaciones entre los nodos se pueden desarrollar sobre una red `Giga-Ethernet`, `Fast Ethernet` o bien, sólo en el caso de los nodos de trabajo, sobre una red de comunicación `Myrinet` a 2.0+2.0 Gbit/s. El sistema operativo del servidor y los nodos de trabajo es Linux, contando con el compilador `gcc` e `icc`.

Esta máquina fue utilizada para obtener los resultados mostrados en las figuras 4.5 (p. 160), 4.30 (p. 208), 4.31 (p. 210), 4.36 (p. 220), 4.37 (p. 224) y 4.40 (p. 228).

A.2.6. `Cluster UJI-SPINE`

El cluster de nombre lógico `spine` se localiza en el PSCOM de la UJI y está compuesto por 32 nodos, cada uno con un procesador Intel Pentium4 a 1.8 GHz con 512 MB RAM, conectados por una red `Myrinet` (1.0 Gb/s) y una red `Fast Ethernet`. El cluster cuenta con un nodo servidor que dispone de un procesador Intel Pentium4 a 2.66 GHz, con 1 GB RAM.

Las figuras 4.3 (p. 157) y 4.7 (p. 164) muestran resultados obtenidos en este cluster.

A.2.7. `Cluster ULL`

El Cluster de nombre lógico `beowulf` del Grupo de Paralelismo de la ULL (PCG-ULL) ha ido modificando sus características, que han ido variando desde sus orígenes como un cluster heterogéneo, formado por 13 PCs con procesadores Pentium II y AMD-K6 y diversas configuraciones dependiendo de cada máquina, hasta el estado actual, en el que el cluster está compuesto

por 48 procesadores Intel Xeon con velocidades que van desde 1.4 hasta 3.2 GHz. Los equipos que forman el cluster contienen uno o varios de estos procesadores (hasta 4) y se distribuyen en 23 nodos, con un rendimiento máximo teórico de 268 Gflops. La memoria total del sistema es de 25 GB y la capacidad de almacenamiento en disco duro es superior a 1.5 TB, siendo el sistema operativo de los equipos Debian GNU Linux 4.0. La interconexión se realiza mediante una red `Infiniband` y dos redes `Gigabit Ethernet`.

En este cluster se tomaron los datos que se muestran en las figuras 4.21 (p. 191), 4.24 (p. 195), 4.33 (p. 214) y 4.34 (p. 216).

A.2.8. `Cray T3E`

El equipo `Cray T3E`, de nombre lógico `crayc` y que perteneció al CIEMAT, es una máquina pura de memoria distribuida, segunda generación de arquitecturas de supercomputadoras masivamente paralelas desarrolladas por el grupo `Cray`. Estaba compuesta por procesadores DEC 21164 (Alpha EV-5) a 300 MHz, con una potencia de pico de 600 Mflops por procesador, con un juego de instrucciones RISC de 64 bits. Cada procesador disponía de 128 MB de memoria distribuida, escalable hasta 2 GB. Contaba con disco duro SCSI de unos 130 GB de capacidad, una red toroidal 3D de interconexión de baja latencia y sistema de E/S paralela basada en la tecnología `GigaRing` de SGI/Cray.

Este equipo se encuentra actualmente descatalogado y fuera de uso. No obstante, realizamos un uso intensivo del mismo durante su periodo de vigencia, ya que era una de las pocas computadoras donde se nos garantizaban ejecuciones en exclusiva, además de que ofrecía 34 procesadores, una de las mayores configuraciones a la que tuvimos acceso.

Esta máquina fue empleada en la toma de los resultados mostrados en las figuras 4.1 (p. 153), 4.10 (p. 167), 4.11 (p. 169), 4.12 (p. 172), 4.15 (p. 176), 4.25 (p. 197), 4.27 (p. 203), 4.28 (p. 204) y 4.36 (p. 220).

A.2.9. `IBM RS-6000`

El sistema IBM RS-6000 cuenta con 8×16 procesadores Nighthawk Power3 a 375 MHz (192 Gflops), 64 GB de memoria y una red `SP Switch2` a 500 Mb/seg. Su nombre lógico es `kadesh` y pertenece al CEPBA de la UPC.

En este equipo se tomaron los resultados computacionales que se muestran en las figuras 4.18 (p. 183) y 4.41 (p. 232).

A.2.10. SGI Altix 250

El sistema multiprocesador SGI Altix 250 del PSCO de la UJI tiene como nombre lógico set y cuenta con 16 procesadores Itanium2 a 1.5 GHz y memoria compartida total de 32 GB. El sistema operativo es Linux, con compiladores gcc e icc.

Los resultados computacionales obtenidos en esta máquina pueden observarse en las figuras 4.44 (p. 239) y 4.46 (p. 242).

A.2.11. SGI Origin 2000 (SGI O2000)

La SGI Origin 2000, de nombre lógico karnak, pertenece al CEPBA de la UPC. Esta máquina tipo cc-NUMA cuenta con 64 procesadores MIPS R10000 a 250 MHz, siendo el máximo pico de rendimiento teórico de 32 Gflops/s. La memoria principal es de 12 GB y dispone de 350 GB de almacenamiento en disco duro.

Los resultados computacionales obtenidos con esta máquina se muestran en las figuras 4.11 (p. 169), 4.14 (p. 174) y 4.36 (p. 220).

A.2.12. SGI Origin 3000 (SGI O3000)

El uso de sistemas de tipo cc-NUMA se ve complementado por la SGI Origin 3000 (serie 3800), perteneciente al CIEMAT, con nombre lógico jen50, que cuenta con 122 procesadores MIPS R14000 a 600 MHz, 160 GB de memoria y sistema operativo IRIX.

Las figuras 4.23 (p. 195), 4.11 (p. 169), 4.21 (p. 191), 4.29 (p. 207) 4.32 (p. 213) y 4.36 (p. 220) recogen resultados computacionales obtenidos con esta máquina.

A.2.13. SunFire 6800

La máquina SunFire 6800 está localizada en el EPCC y su nombre lógico es lomond. Se trata de un cluster HPC Sun con sistema *front-end* SMP HPC 3500 con 8 procesadores UltraSPARC-II a 400 Mhz y 8 GB de memoria compartida, y un *back-end* formado por dos sistemas SMP SunFire 6800 con 24 procesadores UltraSPARC-III a 750 Mhz y un total de 48 GB de memoria compartida. El sistema operativo es Solaris 2.8.

Las siguientes figuras presentan resultados computacionales tomados en esta máquina: 4.2 (p. 155), 4.6 (p. 161), 4.11 (p. 169), 4.13 (p. 172), 4.16 (p. 178), 4.26 (p. 198) y 4.40 (p. 228).

A.2.14. `SunFire E15000` (`SunFire E15K`)

El equipo `SunFire E15000` sucedió al cluster `SunFire 6800` en el EPCC. Esta máquina heredó el nombre lógico `lomond` de su antecesora y cuenta con un *front-end* de 4 procesadores y un *back-end* de 48 procesadores UltraSPARC-III a 900 Mhz. El sistema operativo es Solaris 2.9.

En este sistema se obtuvieron los resultados computacionales que se presentan en la figura 4.37 (p. 224).

A.3. Cuadro Resumen

Para una fácil identificación y consulta del correspondiente equipo en el que se obtuvieron los resultados computacionales, la tabla A.3 (página 260) resume las características más notables de los sistemas utilizados. Los campos de esta tabla son:

1. **Descripción**: Identificación del equipo, indicando nombre del fabricante, modelo o una etiqueta para el caso de los clusters.
2. **N. lógico**: Nombre lógico de la máquina
3. **Tipo procesadores**: Marca, modelo y velocidad de los procesadores
4. **N Procs.**: Total de procesadores de la máquina. Entre paréntesis se especifica el máximo de procesadores que se usó en las ejecuciones, valor que generalmente viene limitado por la topología del equipo, política de particiones, restricciones de las colas de ejecución, reserva de procesadores para gestión, etc. Si un asterisco acompaña a este valor, indica que en ese equipo fue posible obtener la mayoría de resultados computacionales ejecutando en exclusiva, lo que representa una mayor fiabilidad en los tiempos y rendimientos medidos.

Descripción	N. lógico	Tipo procesadores	N. Procs.
Bull NovaScale Server	tarja	Intel-Itanium 2 (1.5GHz)	64 (32*)
Cluster EPCC	ness	AMD Opteron (AMD64e) (2.6 GHz)	2+32 (16)
Cluster IAC	chimera	Intel Xeon (2.8GHz/3.2GHz)	2×32 (32*)
Cluster UJI-CAT	cat	tetraprocesadores Intel Itanium2 (1.5 GHz)	(1+9)×4 (16*)
Cluster UJI-RA	ra	biprocesadores Intel Xeon (2.4 GHz)	(1+34)×2 (32*)
Cluster UJI-SPINE	spine	Intel Pentium4 (1.8 GHz)	32 (14*)
Cluster ULL	beowulf	Intel Xeon (1.4-3.2 GHz)	48 (22*)
Cray T3E	crayc	DEC 21164 (Alpha EV-5) (300 MHz)	34 (34*)
IBM RS-6000	kadesh	Nighthawk Power3 (375 MHz)	8×16 (32)
SGI Altix 250	set	procesadores Itanium2 (1.5 GHz)	16 (16*)
SGI Origin 2000 (SGI O2000)	karnak	R10000 (250 MHz)	64 (32)
SGI Origin 3000 (SGI O3000)	jen50	R14000 (600 MHz)	122 (16)
SunFire 6800	lomond	UltraSPARC-III (750 Mhz)	8+24 (24)
SunFire E15000 (SunFire E15K)	lomond	UltraSPARC-III (900 Mhz)	4+48 (16)

Tabla A.1: Resumen de las principales características de los equipos utilizados para la toma de resultados computacionales

Bibliografía

[1] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[2] Allen, E., Chase, D., Luchangco, V., Maessen, J., Ryu, S., Steele, G., and Tobon-Hochstadt. Fortress language specification v. 1.0. Tech. rep., Sun MicroSystems, Inc., 2008. `research.sun.com/projects/plrg/Publications/fortress.1.0.pdf`.

[3] Almeida, F. Computación en paralelo y sistemas heterogéneos. Entidad Financiadora: Comisión Interministerial de Ciencia y Tecnología Participantes: Dpto. de Estadística, I.O. y Computación de la Universidad de La Laguna, Dpto. de Ingeniería y Ciencia de Computadores de la Universidad Jaime I de Castellón, 2005. Número de Participantes: 21 investigadores.

[4] Almeida, F., Blanco, V., Dorta, A. J., Dorta, I., González, J. A., González, D., León, C., Moreno, L. M., Rodríguez, C., and de Sande, F. Parallel skeletons using `llc`. In *XIV Jornadas de Paralelismo* (Leganés, Madrid, 15–17 de septiembre 2003), pp. 27–32.

[5] Arapov, D., Kalinov, A., Lastovetsky, A., and Ledovskih, I. Experiments with mpC: Efficient solving regular problems on heterogeneous networks of computers via irregularizatio. In *Proceedings of the Fifth International Symposium on Solving Irregularly Structured Problems in Parallel (IRREGULAR'98)* (Berkley, California, USA, 1998), Springer-Verlag, Lecture Notes in Computer Science 1457, pp. 332–343.

[6] Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., Patterson, D. A., Plishker, W. L., Shalf, J., Williams, S. W., and Yelick, K. A. The

landscape of parallel computing research: A view from Berkeley. Tech. Rep. UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[7] Ashby, J. V. New languages for high performance, high productivity computing. Tech. Rep. RAL-TR-2007-012, Computational Science and Engineering Department, STFC Rutherford Appleton Laboratory, August 2007. http://epubs.cclrc.ac.uk/bitstream/1771/new-languages1.pdf.

[8] Bailey, D. H. FFTs in external or hierarchical memory. *The Journal of Supercomputing 4*, 1 (1990), 23–35.

[9] Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Dagum, D., Fatoohi, R. A., Frederickson, P. O., Lasinski, T. A., Schreiber, R. S., Simon, H. D., Venkatakrishnan, V., and Weeratunga, S. K. The NAS parallel benchmarks. *The International Journal of Supercomputer Applications 5*, 3 (Fall 1991), 63–73.

[10] Barriuso, R., and Knies, A. SHMEM user's guide for C. Tech. Rep. SN-2516, Cray Research Inc., June 1994.

[11] Benoit, A., Cole, M., Hillston, J., and Gilmore, S. Flexible skeletal programming with eSkel. In *Proc. of the 11th International Euro-Par Conference* (Lisbon, August 2005), vol. 3648 of *LNCS*, Springer Verlag, pp. 761–770.

[12] Bientinesi, P., Gunnels, J. A., Myers, M. E., Quintana-Ortí, E. S., and van de Geijn, R. A. The science of deriving dense linear algebra algorithms. *ACM Trans. on Mathematical Software 31*, 1 (2005), 1–26.

[13] Blandford, R. D., and Narayan, R. Cosmological applications of gravitational lensing. *Ann. Rev. Astron. Astrophys. 30* (1992), 311–358.

[14] Blikberg, R., and Sørevik, T. Load balancing and OpenMP implementation of nested parallelism. *Parallel Computing 31*, 10-12 (2005), 984–998.

[15] Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. Cilk: An efficient

multithreaded runtime system. *Journal of Parallel and Distributed Computing 37*, 1 (1996), 55–69.

[16] Blumofe, R. D., and Leiserson, C. E. Scheduling multithreaded computations by work stealing. *J. ACM 46*, 5 (1999), 720–748.

[17] Board, O. A. R. *OpenMP C/C++ Application Program Interface*, March 2002. `http://www.openmp.org/specs/mp-documents/cspec20.pdf`.

[18] Bonachea, D. GASNet specification. Tech. rep., Berkeley, CA, USA. `http://gasnet.cs.berkeley.edu`.

[19] Butler, R. M., and Lusk, E. L. Monitors, messages, and clusters: the P4 parallel programming system. *Parallel Computing 20*, 4 (1994), 547–564.

[20] Bykat, A. Convex hull of a finite set of points in two dimensions. *Informations Processing Letters 7* (1978), 296–298.

[21] C2c and Cilk2c Home Page. `http://supertech.lcs.mit.edu/cilk/home/software.html`.

[22] Callahan, D., Chamberlain, B. L., and Zima., H. P. The cascade high productivity language. In *Proc. of the 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2004)* (Los Alamitos, CA, USA, 2004), vol. 00, IEEE Computer Society, pp. 52–60.

[23] Centro Europeo de Paralelismo de Barcelona (CEPBA). `http://www.cepba.upc.es`.

[24] Chamberlain, B., Callahan, D., and Zima, H. Parallel programmability and the Chapel language. *Int. J. High Perform. Comput. Appl. 21*, 3 (2007), 291–312.

[25] Chamberlain, B. L. *The design and implementation of a region-based parallel programming language.* PhD thesis, 2001. Chair-Lawrence Snyder.

[26] Chamberlain, B. L., Choi, S.-E., Lewis, E. C., Lin, C., Snyder, L., and Weathersby, W. D. ZPL's WYSIWYG performance model. In *IEEE Workshop on High-Level Parallel Programming Models and Supportive Environments* (1998).

[27] Chamberlain, B. L., Choi, S.-E., Lewis, E. C., Snyder, L., Weathersby, W. D., and Lin, C. The case for high-level parallel programming in ZPL. *IEEE Comput. Sci. Eng. 5*, 3 (1998), 76–86.

[28] Chandra, R., Menon, R., Dagum, L., Kohr, D., Maydan, D., and McDonald, J. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers. Academic Press, 2001.

[29] Chapiro, D. M. *Globally-Asynchronous Locally-Synchronous Systems*. PhD thesis, Stanford University, October 1984.

[30] Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., and Sarkar, V. X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Not. 40*, 10 (2005), 519–538.

[31] Chen, W.-Y., Iancu, C., and Yelick, K. Communication optimizations for fine-grained UPC applications. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 267–278.

[32] Centro de Investigaciones Energéticas, Medioambientales y Tecnológicas (CIEMAT). `http://www.ciemat.es`.

[33] The Cilk Project. `http://supertech.lcs.mit.edu/cilk`.

[34] Cole, M. *Algorithmic Skeletons: structured management of parallel computation*. Monograms. Pitman/MIT Press, Cambridge, MA, 1989.

[35] Cole, M. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Comput. 30*, 3 (2004), 389–406.

[36] Cole, M. I. *The eSkel Project*. Division of Informatics, University of Edinburgh. `http://www.dcs.ed.ac.uk/home/mic/eSkel/`.

[37] COMPunity. `http://www.compunity.org/`.

[38] Cooley, J. W., and Tukey, J. W. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation 19*, 90 (April 1965).

[39] Cormen, T. H., Leiserson, C. E., , and Rivest, R. L. *Introduction to Algorithms*. MIT Press and McGraw Hill, 1990.

[40] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms*, 2nd ed. The MIT Press, 2001.

[41] Cxref Home Page. `http://www.gedanken.demon.co.uk/cxref/`.

[42] Dagum, L., and Menon, R. OpenMP: An industry-standard API for shared-memory programming. *IEEE Comput. Sci. Eng. 5*, 1 (1998), 46–55.

[43] The Defense Advanced Research Projects Agency (DARPA). `http://www.darpa.mil/`.

[44] de Sande, F. *El modelo de Computación Colectiva: Una metodología eficiente para la ampliación del modelo de librería de paso de mensajes con paralelismo anidado.* PhD thesis, Universidad de La Laguna, La Laguna, December 1998.

[45] de Sande, F. LLASKEL: Una aproximación a la programación paralela estructurada basada en esqueletos. Entidad Financiadora: Dirección General de Universidades. Gobierno de Canarias., 2004. Publicado en el BOC del lunes 26 de enero de 2004, pg. 920.

[46] `DIMEMAS`: Utilidad de análisis de rendimiento para aplicaciones de paso de mensajes. `http://www.cepba.upc.es/dimemas/`.

[47] Dinda, P., Gross, T., Hallaron, D. O., Segall, E., Stichnoth, J., Subhlok, J., Webb, J., and Yang, B. The CMU task parallel program suite. Tech. Rep. CMU-CS-94-131, School of Computer Science, Carnegie Mellon University, 1994.

[48] Dongarra, J. J., Croz, J. D., Hammarling, S., and Hanson, R. J. An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software 14*, 1 (1988), 1–17.

[49] Dongarra, J. J., Meuer, H. W., and Strohmaier, E. Top500 supercomputer sites, 11th edition. Tech. Rep. UT-CS-98-391, 1998.

[50] Dorta, A. J., Badía, J. M., Quintana, E. S., and de Sande, F. Implementing OpenMP for clusters on top of MPI. In *Proc. of the 12th European PVM/MPI Users' Group Meeting* (Sorrento, Italy, September 18–21 2005), vol. 3666 of *LNCS*, Springer-Verlag, pp. 148–155.

[51] Dorta, A. J., Badía, J. M., Quintana, E. S., and de Sande, F. Parallelizing dense linear algebra operations with task queues in `llc`. In *Proc. of the 14th European PVM/MPI Users' Group Meeting* (Paris, France, 2007), Lecture Notes in Computer Science, Springer-Verlag.

[52] Dorta, A. J., de Sande, F., Badía, J. M., and Quintana, E. S. OpenMP para clusters. In *XVI Jornadas de Paralelismo* (Granada, 13–16 de septiembre 2005).

[53] Dorta, A. J., de Sande, F., Badía, J. M., and Quintana, E. S. Paralelización de operaciones de álgebra lineal densa en `llc`. In *XVIII Jornadas de Paralelismo* (Zaragoza, 11–14 de septiembre 2007).

[54] Dorta, A. J., García, L., González, J. R., León, C., and Rodríguez, C. Aproximación paralela a la técnica divide y vencerás. In *Proc. de las X Jornadas de Enseñanza Universitaria de la Informática (JENUI'2004)* (Alicante, 14–16 de julio 2004), pp. 371–378.

[55] Dorta, A. J., González, J. A., Rodríguez, C., and de Sande, F. Towards structured parallel programming. In *Proc. Fourth European Workshop on OpenMP (EWOMP 2002)* (Rome, Italy, September 2002).

[56] Dorta, A. J., González, J. A., Rodríguez, C., and de Sande, F. `llc`: A parallel skeletal language. *Parallel Processing Letters 13*, 3 (September 2003), 437–448.

[57] Dorta, A. J., González-Escribano, A., Rodríguez, C., and de Sande, F. The OpenMP source code repository: an infrastructure to contribute to the development of OpenMP. In *Proc. of the 6th European Workshop on OpenMP (EWOMP 2004)* (Stockholm, Sweden, October 2004), pp. 57–62.

[58] Dorta, A. J., González-Escribano, A., Rodríguez, C., and de Sande, F. `OmpSCR`: una infraestructura para contribuir al desarrollo de OpenMP. In *XV Jornadas de Paralelismo* (Almería, 15–17 de septiembre 2004).

[59] Dorta, A. J., González-Escribano, A., Rodríguez, C., and de Sande, F. The OpenMP source code repository. In *Proc. of the 13th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP 2005)* (Lugano, Switzerland, February 2005), pp. 244–250.

[60] Dorta, A. J., Lopez, P., and de Sande, F. Basic skeletons in `llc`. *Parallel Computing 32*, 7–8 (September 2006), 491–506.

[61] Dotsenko, Y., Coarfa, C., and Mellor-Crummey, J. A multi-platform co-array Fortran compiler. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 29–40.

[62] El-Ghazawi, T., Carlson, W., Sterling, T., and Yelick, K. *UPC: Distributed Shared Memory Programming.* Wiley-Interscience, 2005.

[63] El-Ghazawi, T., and Smith, L. UPC: unified parallel C. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing* (New York, NY, USA, 2006), ACM, p. 27.

[64] Edinburgh Parallel Computing Center (EPCC). `http://www.epcc.ed.ac.uk`.

[65] Extensions to the C Language Family. `http://gcc.gnu.org/onlinedocs/gcc/C-Extensions.html`.

[66] Fagg, G., and Dongarra, J. PVMPI: An integration of PVM and MPI systems. *Calculateurs Parallèles 8*, 2 (1996), 151–166.

[67] Forte, L. G. La laguna C, una herramienta para el desarrollo de algoritmos con paralelismo anidado. Tech. rep., 1997.

[68] Fractal del Conjunto de Mandelbrot en la Wikipedia. `http://es.wikipedia.org/wiki/Conjunto_de_Mandelbrot`.

[69] Frigo, M., Leiserson, C. E., and Randall, K. H. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)* (17–19 June, Montreal, Canada, 1998), pp. 212–223.

[70] FSF-GNU. Bison GNU parser generator. `http://www.gnu.org/software/bison/`.

[71] FSF-GNU. Flex, the fast lexical analyzer. `http://www.gnu.org/software/flex/`.

[72] Global Arrays. `http://www.emsl.pnl.gov/docs/global/`.

[73] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V. S. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. Scientific and engineering computation. MIT Press, Cambridge, MA, USA, 1994.

[74] Geist, G. A., Kohla, J. A., and Papadopoulos, P. M. PVM and MPI: A Comparison of Features. *Calculateurs Paralleles 8*, 2 (1996), 137–150.

[75] Goicoechea, L. J., Alcalde, D., Mediavilla, E., and Muñoz, J. A. Determination of the properties of the central engine in microlensed QSOs. *Astronomy and Astrophysics 397* (2003), 517–525.

[76] Golub, G. H., and O'Leary, D. P. Some history of the conjugate gradient and Lanczos methods. *SIAM Review 31* (1989), 50–102.

[77] Golub, G. H., and Van Loan, C. F. *Matrix Computations*, third ed. Johns Hopkins University Press, Baltimore, MD, 1996.

[78] Gonzalez, M., Ayguade, E., Martorell, X., and Labarta, J. Exploiting pipelined executions in OpenMP. In *Procs. International Conference on Parallel Processing (ICPP'03)* (Los Alamitos, CA, USA, 2003), vol. 00, IEEE Computer Society, p. 153.

[79] Gonzalez, M., Oliver, J., Martorell, X., Ayguade, E., Labarta, J., and Navarro, N. OpenMP extensions for thread groups and their run-time support. In *Proc. of the Workshop on Languages and Compilers for Parallel Computing* (2001), vol. 2017 of *LNCS*, pp. 324–338.

[80] Hadjidoukas, P. E., Polychronopoulos, E. D., and Papatheodorou, T. S. A modular OpenMP implementation for clusters of multiprocessors. *Journal of Parallel and Distributed Computing Practices (PDCP), Special Issue on OpenMP: Experiences, Implementations and Applications 2*, 5 (2004), 153–168.

[81] Hageman, L. A., and Young, D. M. *Applied Iterative Methods*. Academic Press, New York, 1981.

[82] Hilfinger, P.Ñ., Bonachea, D. O., Datta, K., Gay, D., Graham, S. L., Liblit, B. R., Pike, G., Su, J. Z., and Yelick, K. A. Titanium language reference manual. Tech. Rep. UCB/EECS-2005-15, University of California at Berkeley, November 2005.

[83] Hoare, C. A. R. ACM Algorithm 64: Quicksort. *Communications of the ACM 4*, 7 (July 1961), 321.

[84] HPC-Europa Transnational Access Programme. `http://www.hpc-europa.org`.

[85] Huang, L., Chapman, B., and Liu, Z. Towards a more efficient implementation of OpenMP for clusters via translation to Global Arrays. Tech. Rep. UH-CS-04-05, Department of Computer Science, Univeristy of Houston, December 2004.

[86] Husbands, P., Iancu, C., and Yelick, K. A performance analysis of the Berkeley UPC compiler. In *ICS '03: Proceedings of the 17th annual international conference on Supercomputing* (New York, NY, USA, 2003), ACM, pp. 63–73.

[87] Instituto de Astrofísica de Canarias (IAC). `http://www.iac.es`.

[88] Inc., C. Chapel specification 4.0. Tech. rep., Cray Inc., 2005. `http://chapel.cs.washington.edu/specification.pdf`.

[89] Jin, H., Frumkin, M., and Yan, J. The OpenMP implementation of NAS parallel benchmarks and its performance. Technical Report NAS-99-011, NASA Ames Research Center, 1999.

[90] Johnson, S. C. YACC: Yet another compiler compiler. In *UNIX Programmer's Manual*, vol. 2. Holt, Rinehart, and Winston, New York, NY, USA, 1979, pp. 353–387.

[91] Kennedy, K., and Allen, J. R. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[92] Kennedy, K., Koelbel, C., and Zima, H. The rise and fall of High Performance Fortran: an historical object lesson. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages* (New York, NY, USA, 2007), ACM, pp. 7–1–7–22.

[93] Knuth, D. E. *The art of computer programming / Donald E. Knuth*. Addison-Wiley, Reading, Mass., 1968.

[94] Koelbel, C. H., Loveman, D. B., Schreiber, R. S., Guy L. Steele, J., and Zosel, M. E. *The High Performance Fortran handbook*. MIT Press, Cambridge, MA, USA, 1994.

[95] Kumar, V., Grama, A., Gupta, A., and Karypis, G. *Introduction to Parallel Computing: Design and Analysis of Algorithms.* Benjamin Cummings, Redwood City, CA, 1994.

[96] Lawson, C. L., Hanson, R. J., Kincaid, D. R., and Krogh, F. T. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Softw. 5*, 3 (1979), 308–323.

[97] Lee, J. ANSI C draft: Lex specification and YACC grammar. Tech. rep., 1985. `ftp.uu.net,usenet/net.sources/ansi.c.grammar.Z`.

[98] León Hernández, C. *Diseño e implementación de lenguajes orientados al modelo PRAM.* PhD thesis, Universidad de La Laguna, La Laguna, 1996.

[99] Lewis, J. G., and van de Geijn, R. A. Distributed memory matrix-vector multiplication and conjugate gradient algorithms. In *Supercomputing '93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing* (New York, NY, USA, 1993), ACM, pp. 484–492.

[100] `llc` Home Page. `http://llc.pcg.ull.es`.

[101] Loan, C. F. V. *Computational Frameworks for the Fast Fourier Transform.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1992.

[102] Lopez, J. L. llcc98: Una herramienta eficiente para el desarrollo de algoritmos paralelos. Tech. rep., 1998.

[103] López, P., Dorta, A. J., Mediavilla, E., and de Sande, F. Generation of microlensing magnification patterns with high performance computing techniques. In *Workshop on state-of-the-art in Scientific and Parallel Computing 9th International Conference, PARA 2006* (Umea, Sweden, 2006), Lecture Notes in Computer Science.

[104] Lusk, E., and Yelick, K. Languages for high-productivity computing: The DARPA HPCS language project. *Parallel Processing Letters 17*, 1 (2007), 89–102. `doi:10.1142/S0129626407002892`.

[105] Mainwaring, A., and Culler, D. Generic active message interface specification v1.1. In *U.C. Berkeley Computer Science Technical Report* (February 1995).

[106] Mainwaring, A. M., and Culler, D. E. Active message applications programming interface and communication subsystem organization. Tech. Rep. UCB/CSD-96-918, EECS Department, University of California, Berkeley, October 1996.

[107] Mandelbrot, B. B. *Form, chance, and dimension. Translation of Les objets fractals: Partition*. W. H. Freeman, San Francisco, 1977.

[108] Mandelbrot, B. B. *The Fractal Geometry of Nature*. Freeman and Co., San Francisco, San Francisco, 1982.

[109] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. University of Tennessee, Knoxville, TN, 1995. `http://www.mpi-forum.org/`.

[110] Miller, R. C. *A Type-checking Preprocessor for Cilk 2 a Multithreaded C Language*. PhD thesis, Massachusetts Institute of Technology, 1995. `ftp://theory.lcs.mit.edu/pub/cilk/rcm-msthesis.ps.Z`.

[111] Morales, D., Almeida, F., García, F., Roda, J. L., and Rodríguez, C. Design of parallel algorithms for the single resource allocation problem. *European Journal of Operational Research 126*, 1 (October 2000), 166–174.

[112] NAS Parallel Benchmarks. `http://www.nas.nasa.gov/NAS/NPB/`.

[113] NASA advanced supercomputing division. `http://www.nas.nasa.gov/`.

[114] Nieplocha, J., and Carpenter, B. ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. 533–546.

[115] Nieplocha, J., Palmer, B., Tipparaju, V., Krishnan, M., Trease, H., and Aprà, E. Advances, applications and performance of the global arrays shared memory programming toolkit. *Int. J. High Perform. Comput. Appl. 20*, 2 (2006), 203–231.

[116] Nitsche, T. Lifting sequential functions to parallel skeletons. *Parallel Processing Letters 12*, 2 (2002), 267–284.

[117] Numrich, R. W., and Reid, J. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum 17*, 2 (1998), 1–31.

[118] Numrich, R. W., and Reid, J. Co-arrays in the next Fortran standard. *SIGPLAN Fortran Forum 24*, 2 (2005), 4–17.

[119] Okutomi, M., and Kanade, T. A multiple-baseline stereo. *IEEE Transactions on Pattern Analysis and Machine Intelligence 15*, 4 (1993), 353–63.

[120] *Omni OpenMP Compiler Project*. `http://phase.etl.go.jp/Omni/`.

[121] `OmpSCR` en `sourceforge.net`. `http://sourceforge.net/projects/ompscr/`.

[122] OpenMP Architecture Review Board. OpenMP official web site. `http://www.openmp.org/`.

[123] OpenMP Architecture Review Board. *OpenMP Application Program Interface v. 2.5*, May 2005. `http://www.openmp.org/drupal/mp-documents/spec25.pdf`.

[124] OpenMP Architecture Review Board. *OpenMP Application Program Interface v. 3.0*, May 2008. `http://www.openmp.org/drupal/mp-documents/spec30.pdf`.

[125] `PARAVER`: Utilidad flexible de análisis y visualización de rendimiento. `http://www.cepba.upc.es/paraver/`.

[126] Parzyszek, K., Nieplocha, J., and Kendall, R. A. A generalized portable SHMEM library for high performance computing. In *Twelfth IASTED International Conference on Parallel and Distributed Computing and Systems* (Las Vegas, Nevada, 2000), M. Guizani and Z. Shen, Eds., IASTED /Acta Press, pp. 401–406. ISSN: 0-88986-266-4.

[127] Grupo de Computación Paralela (PCG) de la Universidad de La Laguna (ULL). `http://www.pcg.ull.es`.

[128] Pike, G., Semenzato, L., Colella, P., and Hilfinger, P.Ñ. Parallel 3D adaptive mesh refinement in Titanium. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing* (San Antonio, Texas, March 1999).

[129] Grupo de Computación Científica Paralela de la Universidad Jaume I de Castellón. `http://www.pscom.uji.es`.

[130] Quinn, M. J. *Parallel computing (2nd ed.): theory and practice.* McGraw-Hill, Inc., New York, NY, USA, 1994.

[131] Quinn, M. J. *Parallel Programming in C with MPI and OpenMP.* McGraw-Hill Education Group, 2003.

[132] Quintana, E. S. Red de computación de altas prestaciones sobre arquitecturas paralelas heterogéneas (CAPAP-H). Entidad Financiadora: Ministerio de Educación y Ciencia, Acción Complementaria, 2008–2009.

[133] Quintana, E. S. COPABIB: construcción y optimización automáticas de bibliotecas paralelas de computación científica. Entidad Financiadora: Comision Interministerial de Ciencia y Tecnología Participantes: Universidad de La Laguna, Universidad Jaime I de Castellón, Universidad de Alicante, Universidad de Murcia, 2009. Aprobado, pendiente de su ejecución.

[134] Rodríguez-Rosa, J., Dorta, A. J., Rodríguez, C., and de Sande, F. Exploiting task and data parallelism. In *Proc. of the Fifth European Workshop on OpenMP (EWOMP 2003)* (Aachen, Germany, September 2003), pp. 107–116.

[135] Servicio de Apoyo Informático a la Investigación de la Universidad de La Laguna, 2007. `http://www.saii.ull.es`.

[136] Sato, M., Harada, H., and Hasegawa, A. Cluster-enabled OpenMP: An OpenMP compiler for the SCASH software distributed shared memory system. *Scientific Programming, Special Issue: OpenMP 9*, 2-3 (2001), 123–130.

[137] Schroeder, L. *Buffon's needle problem: An exciting application of many mathematical concepts*, vol. 67. Mathematics Teacher, 1974.

[138] Shah, S., Haab, G., Petersen, P., and Throop, J. Flexible control structures for parallelism in OpenMP. In *1st European Workshop on OpenMP* (Lund, Sweden, September 1999).

[139] Shah, S., Haab, G., Petersen, P., and Throop, J. Flexible control structures for parallelism in OpenMP. *Concurrency: Practice and Experience 12*, 12 (October 2000), 1219–1239.

[140] Snir, M., Otto, S., Huss-Lederman, S., Walker, D., and Dongarra, J. J. *MPI: The Complete Reference.* MIT Press,

Cambridge, MA, 1995.
http://www.netlib.org/utk/papers/mpi-book/mpi-book.html.

[141] Snyder, L. *A programmer's guide to ZPL*. MIT Press, Cambridge, MA, USA, 1999.

[142] Supercomputing Technologies Group, Massachusetts Institute of Technology Laboratory for Computer Science. *Cilk Reference Manual*, Nov. 2001.
http://supertech.csail.mit.edu/cilk/.

[143] Swope, W. C., Andersen, H. C., Berens, P. H., and Wilson, K. R. A computer simulation method for the calculation of equilibrium constants for the formation of physical clusters of molecules: Application to small water clusters. *Journal of Chemical Physics 76* (1982), 637–649.

[144] Terboven, C. First experiments with tasking in OpenMP 3.0, 2008.
http://terboven.spaces.live.com/.

[145] Tian, X., Girkar, M., Shah, S., Armstrong, D., Su, E., and Petersen, P. Compiler support of the workqueuing execution model for Intel SMP architectures. In *Proc. of the Eighth International Workshop on High-Level Parallel Programming Models and Supportive Environments* (Nice, Paris, September 2003), pp. 47–55.

[146] Tuckerman, M., Berne, B. J., and Martyna, G. J. Reversible multiple time scale molecular dynamics. *Journal of Chemical Physics 97*, 3 (1992), 1990–2001.

[147] Universidad Jaime I de Castellón (UJI). http://www.uji.es.

[148] Universidad de La Laguna (ULL). http://www.ull.es.

[149] University, R. High Performance Fortran language specification. *SIGPLAN Fortran Forum 12*, 4 (1993), 1–86.

[150] Universidad Politécnica de Cataluña (UPC). http://www.upc.es.

[151] UPC language specification. The UPC Consortium. http://upc.gwu.edu.

[152] Van Zee, F., Bientinesi, P., Low, T. M., and van de Geijn, R. A. Scalable parallelization of FLAME code via the workqueuing model. *ACM Trans. on Mathematical Software*. To appear. http://www.cs.utexas.edu/users/flame/pubs.html.

[153] Webb, J. A. Implementation and performance of fast parallel multi-baseline stereo vision. In *Proc. of the Computer Architectures for Machine Perception* (New Orleans, LA, USA, Dec 1993), pp. 232–240.

[154] The X10 programming language. `http://x10.sourceforge.net/`.

[155] Yelick, K., Bonachea, D., Chen, W.-Y., Colella, P., Datta, K., Duell, J., Graham, S. L., Hargrove, P., Hilfinger, P., Husbands, P., Iancu, C., Kamil, A., Nishtala, R., Su, J., Welcome, M., and Wen, T. Productivity and performance using partitioned global address space languages. In *PASCO '07: Proceedings of the 2007 international workshop on Parallel symbolic computation* (New York, NY, USA, 2007), ACM, pp. 24–32.

[156] Yelick, K., Semenzato, L., Pike, G., Miyamoto, C., Liblit, B., Krishnamurthy, A., Hilfinger, P., Graham, S., Gay, D., Colella, P., and Aiken, A. Titanium: A high-performance Java dialect. *Concurrency: Practice and Experience 10*, 11–13 (1998), 825–836.

[4] [58] [52] [53] [55] [6]