UNIVERSIDAD DE ALMERÍA

TESIS

Reconstrucción 3D a partir de Proyecciones en Entornos Multihebrados

Directores:

Dr. Javier Roca Piera

Dr. José Jesús Fernández Rodríguez

Autor:

José Antonio Álvarez Bermejo

When you make the finding yourself, even if you are the last person on Earth to see the light, you will never forget it.

Carl Sagan

Agradecimientos

La realización de este trabajo no habría podido llevarse a cabo sin la inestimable colaboración de mis directores de tesis, Javier Roca Piera y José Jesús Fernández Rodríguez, a los que quiero expresar mi enorme agradecimiento por su trabajo de dirección y constante ayuda. En especial a Javier Roca Piera agradecer su paciencia y disposición a debatir ideas e implementaciones, por el apoyo cuando las cosas van mal y el ánimo cuando van bien, por haberme guiado en el inicio de mi etapa de investigación y aconsejarme en mi labor como docente. A José Jesús Fernández Rodríguez por ser el ejemplo a seguir.

A mi mujer, Dolores Mari, porque sin todo su cariño me habrían faltado fuerzas para abordar cada día de trabajo cuando las cosas han ido mal. A mi pequeño (o pequeña) porque desde que sé que estás ahí, todo ha cambiado a mejor.

A mis padres y hermanos, porque gracias a ellos he andado mi senda.

A mis profesores El Amin Kaidi Lhachmi y Manuel Navarro Bernal, por ser quienes me enseñaron a pensar. Sin su paciencia y dedicación quizá no me habría ido tan bien. Y a otros muchos, que me dejo en el tintero pero no dejan de ser importantes para mí.

A Inmaculada García Fernández y con ella a todos mis compañeros del grupo de investigación Supercomputación: Algoritmos que han estado y están a mi lado, por esos momentos de debate enriquecedor junto a otros de revitalizador descanso. A todos mis compañeros del departamento de Arquitectura de Computadores y Electrónica porque nunca me han fallado.

A Paco Guil, Luis Belmonte y otros tantos sin los que la vida en la universidad sería menos agradable.

Acrónimos

Prólogo

ILP Instruction Level Parallelism, hace referencia a la búsqueda del paralelismo implícito en el repertorio de instrucciones. Consiste en reorganizar las instrucciones para poder ejecutar el mayor número posible de ellas por ciclo.

DLP Data Level Parallelism. Hace referencia a la división del dominio del problema en subdominios suceptibles de ser tratados en paralelo. Las aplicaciones donde se da el DLP suelen ser las aplicaciones multimedia y las aplicaciones científicas, aplicaciones que tienden a mostrar un comportamiento iterativo predecible, con bucles cuyas iteraciones son muy costosas en tiempo, que operan sobre grandes conjuntos de datos, con patrones regulares de acceso a los mismos, etc...

FMA Fused Multiply-add, son instrucciones que mezclan sumas y productos. El repertorio de instrucciones FMA es una extensión al repertorio de instrucciones (128bits) SIMD del microprocesador x86.

noFP No Floating Point. Instrucciones no-FP son instrucciones que no usan la circuitería destinada a la aritmética de punto flotante.

NUMA Non Uniform Memory Access.

UMA Uniform Memory Access.

C Lenguaje de programación C.

MPI Message Passing Interface, conjunto de librerías que se usan para que dos o más procesos, siguiendo una topología determinada, puedan intercambiar datos útiles para su computación.

AMPI Adaptive MPI, es una librería de adaptación de aplicaciones que han sido desarrolladas bajo el modelo de trabajo MPI. Se embute en un runtime más complejo denominado Charm.

Capítulo 1

RFC Request for Comments, petición de comentarios. Los RFC publicaciones de las notas o acuerdos tomados sobre ciertos aspectos de la red Internet, el RFC se usa para conocer la opinión de un espectro más amplio de profesionales.

VLSI Very Large Scale of Integration.

ILP Instruction Level Parallelism.

TLP Task Level Parallelism.

VLIW Very Long Instruction Word.

IA-64 Intel Architecture 64 bits.

EPIC Explicitly Parallel Instruction Computing. Son arquitecturas que palian muchos de los defectos de la computación basada en VLIW.

GPU Graphic Processing Unit. Unidad de procesamiento gráfico incorporado en las tarjetas gráficas actuales.

PL Pipe Line Parallelism. Abstracción muy usada para programas en los que se identifican fases o etapas.

OpenMP OpenMP es una interfaz de programación de aplicaciones (API) para la programación multiproceso de memoria compartida en múltiples plataformas. Permite añadir concurrencia a los programas escritos en C, C++ y Fortran sobre la base del modelo de ejecución fork-join. Está disponible en muchas arquitecturas, incluidas las plataformas de

Unix y de Microsoft Windows. Se compone de un conjunto de directivas de compilador, rutinas de biblioteca, y variables de entorno que influencian el comportamiento en tiempo de ejecución.

MIMD Multiple Instructios, Multiple Data. Las máquinas MIMD tienen un número de procesadores que funcionan independientemente y sin sincronización. En cualquier momento, pueden ser diferentes procesadores y diferentes instrucciones sobre la ejecución de diferentes dominios de datos las que se están llevando a cabo.

UMA Uniform Memory Access. Arquitectura en la que el tiempo de acceso a los datos, por todos los procesadores, es el mismo.

NUMA Non Uniform Memory Access. Arquitectura en la que el tiempo de acceso a los datos, por todos los procesadores, no es el mismo.

SIMD Single Instruction, Multiple Data. Es una técnica empleada para conseguir paralelismo a nivel de datos.

MISD Multiple Instructions, Single Data (Múltiples Instrucciones, Un Dato) es un tipo de arquitectura de computación paralela donde muchas unidades funcionales realizan diferentes operaciones en los mismos datos.

SMP Simmetric Multi Processing. Varios microprocesadores comparten el acceso a la memoria. Todos los microprocesadores compiten en igualdad de condiciones por dicho acceso, de ahí la denominación ”simétrico”.

AMD Advanced MicroDevices. Compañía de diseño y fabricación de microprocesadores.

cc-NUMA Cache-Coherent Non Uniform Memory Access. Modelo de arquitectura NUMA donde la coherencia en el acceso a los datos, se mantiene a nivel de cache.

HT HyperThreading, es una marca registrada de la empresa Intel para nombrar su implementación de la tecnología Multithreading Simultáneo también conocido como *SMT*. Permite a los programas preparados para ejecutar múltiples hilos (multi-threaded) procesarlos en paralelo dentro de un único procesador, incrementando el uso de las unidades de ejecución del procesador.

CMP Chip MultiProcessor. Un chip que contiene a más de un núcleo.

API Application Programmers Interface. Conjunto de funciones que ofrecen las librerias o runtimes y que son visibles para el desarrollador.

PThreads En las arquitecturas de memoria compartida (SMP), las hebras se usan para implementar el paralelismo. Cada arquitectura SMP tenía sus propias librerías y APIs para trabajar con hebras. Para los sistemas UNIX se creó una interfaz para trabajar con hebras desde el lenguaje de programación C. Esta librería está descrita en el estándar IEEE POSIX 1003.1c. Las implementaciones que se adhieren a este estándar se denominan pthreads o posix threads.

PVM Parallel Virtual Machine. Es una librería para el cómputo paralelo en un sistema distribuido de procesadores. Está diseñado para permitir que una red de computadoras heterogénea comparta sus recursos de cómputo.

HPF High Performance Fortran. Implementaciones especializadas de Fortran.

UPC Unified Parallel C, extensión del lenguaje de programación C, desarrollada en Berkeley para abordar desarrollos de alto rendimiento.

TLS Thread Local Storage. Almacenamiento Local a una hebra. En computación hebrada uno de los grandes problemas son las variables compartidas, usar el TLS es una buena práctica para evitar los inconvenientes derivados de las variables globales.

IPC Inter Process Communication. Los procesos pueden comunicarse entre sí a través de compartir espacios de memoria, ya sean variables compartidas o buffers, o a través de las herramientas provistas por las rutinas de IPC. La IPC provee un mecanismo que permite a los procesos comunicarse y sincronizarse entre sí, normalmente a través de un sistema de bajo nivel de paso de mensajes que ofrece la red subyacente.

LWP Light Weight Process. Es un medio de conseguir la multitarea. Consiste en (según su uso tradicional en arquitecturas Unix System V y Solaris), que un LWP se ejecuta en el espacio de direcciones de usuario en conexión con una hebra del kernel (sistema operativo) y comparte el espacio de direcciones y otros recursos del sistema con otros LWPs dentro del mismo proceso. De esta forma se pueden mantener varias hebras de usuario , gestionadas a través de una librería de gestión de hebras, sobre uno o varios LWP.

RAM Random Access Memory.

L1 Nivel de cache L1. Suele usarse para almacenar instrucciones bajo el principio de localidad temporal.

L2 Nivel de cache L2. Suele usarse para almacenar datos bajo el principio de localidad espacial.

L3 Nivel de cache L3. Suele emplearse para almacenar datos e instrucciones.

O.O. Orientación a Objetos.

Capítulo 2

3DEM 3D Electron Microscopu

SNR Signal-to-Noise Ratio

HVEM High Voltage Electron Microscopy

CCD Charge-Coupled Device. Circuito integrado que contiene un número determinado de condensadores enlazados o acoplados.

WBP Weighted Back Projection.

SIRT Simultaneous Iterative Reconstruction Technique.

ART Algebraic Reconstruction Technique.

SART Simultaneous Algebraic Reconstruction Technique.

CAV Componente Averaging.

BICAV Block Iterative Component Averaging Methods.

Capítulo 3

MPICH MPICH es una librería de desarrollo de libre disposición de MPI, una norma estándar de paso de mensaje para aplicaciones de memoria distribuida que utilizan computación paralela.

MPILAM LAM/MPI es una implementación open-source de la especificación MPI.

Open MPI Open MPI es una implementación open source de MPI-2.

HPC High Performance Computing. El campo de computación de alto rendimiento es una herramienta muy importante en el desarrollo de simulaciones computacionales a problemas complejos. Para lograr este objetivo, la computación de alto rendimiento se apoya en tecnologías computacionales como los clusters, supercomputadores o mediante el uso de la computación paralela.

CSP Communicating Sequential Processes. Lenguaje formal para describir patrones de interacción en sistemas concurrentes. Forma parte de la familia de teorías matemáticas sobre concurrencia conocias como álgebras de proceso.

CTJ Communicating Threads for Java. Implementación de CSP particularizada a Java. Hace uso de un concepto novedoso y es el del uso de canales de comunicación entre hebras.

MMU Memory Management Unit, algunas veces denominada paged memory management unit (PMMU), es un componente hardware responsable de gestionar las peticiones que la CPU hace a la memoria.

IOS Internet Operating System.

RTS Runtime System.

VP Virtual Processor. Consiste en usar hebras de usuario capaces de empaquetar procesos convencionales de modo que el procesa pueda ser gestionado como si fuera una hebra.

TSD Thread Specific Data. Las hebras operan en áreas de memoria compartida. En ocasiones interesa que la hebra sea la única propietaria de ciertos datos. Para ello se procede a la copia de este dato en un área específicia y de acceso exclusivo a la hebra.

GOT Global Object Table. Sección de un ejecutable donde poder acceder a la lista de variables globales. Es posible hacer transparente el uso de las variables globales si cada hebra almacena en su TSD, de manera transparente, el GOT.

ELF Executable and Linkable Format.

PE Processing Entity o Processing Element.

NCSA National Centre for SuperComputing Applications.

ECC Error Correction Codes. Sistema que usan algunas RAM para corregir posibles errores.

DDR Double Data Rate.

SRAM Static RAM

GPGPU General Purpose Graphics Processing Unit.

PO Parallel Object.

CPAN Construcción Paralela de Alto Nivel.

HLPC High Level Parallel Composition.

MAXPAR Paradigma donde todos los componentes de una aplicación trabajan con el máximo grado de concurrencia posible.

CCS Client-Connect Server.

FSB Front Side Bus.

JVM Java Virtual Machine.

Capítulo 4

FIFO First In, First Out.

PerfVis Performance Visualizer.

Índice general

Índice de figuras

Índice de Tablas

Prólogo

0.1. Background

La computación científica ha alcanzado en los últimos años, con la aparición de la plataforma multicore, una complejidad y una sofisticación sin precedentes, todo esto debido al aumento de capacidad computacional aparentemente ilimitada. Tanto la complejidad de los algoritmos como la amplia configuración de entornos hardware y la cada vez más creciente demanda de software modular, portable y tolerante a fallos han provocado el desarrollo de líneas de trabajo basadas en incluir abstracciones en los lenguajes de alto nivel (e incluso en capas de bajo nivel donde es más difícil pensar en abstracciones [2]), procurando que esto no suponga una penalización para el rendimiento [3].

Habitualmente, la aplicación que se llevará a una arquitectura paralela pertenece a un campo científico ajeno en gran parte al programador encargado de realizar el traslado de ese modelo computacional a una máquina paralela. La descomposición de datos y de computación que se lleve a cabo puede afectar de manera muy negativa al rendimiento de la aplicación si no se realiza de manera adecuada (un ejemplo particular de esto puede verse en [4] donde los autores mejoran el rendimiento de la aplicación reestructurando la forma en que se reparten los datos). Incluso el propio modelo computacional escogido e implementado puede no ser el acertado y además dar lugar a una aplicación poco eficiente, puede lastrar su posterior paralelización.

Es común que sea un grupo científico el que explique con un algoritmo cómo ha de funcionar la aplicación (por ejemplo, a partir del modelo matemático expresado en LaTeX [5]). El traslado del algoritmo a un modelo computacional suele ser realizado, en muchas ocasiones, por el mismo grupo de trabajo (científicos no especialistas en computación), usando frecuentemente

lenguajes poco apropiados [6]. Cada vez se constata con mayor profundidad la necesidad de construir equipos multidisciplinares en los cuales se trabaje de forma conjunta para trasladar el problema científico a la solución computacional aplicada a una determinada arquitectura. El modelo computacional creado, en este último caso, está en manos del programador y en cómo ha conseguido comprender los detalles del algoritmo y del fenómeno a simular.

La creación de estructuras de datos [7] así como la disciplina en el desarrollo pueden ser factores determinantes en la utilidad de la aplicación. La aplicación posteriormente será portada a una arquitectura paralela (o a varias) teniendo que elaborar diferentes estrategias en función de la arquitectura en la que se realizará el despliegue de la aplicación. Y en la mayoría de los casos se hace preciso, incluso, la reescritura del primer modelo computacional para replantear su ejecución en una arquitectura paralela determinada.

A la par que entramos en la era multicore y en la etapa de los petaflops, estamos experimentando un crecimiento sin precedentes en el uso de aceleradores computacionales, arquitecturas heterogéneas, unidades gráficas de procesamiento (GPU), las GPU de propósito general [4], aceleradores de operaciones con puntos flotantes (Clearspeed), arquitecturas especiales como la Cell, de IBM, entre otros muchos [8]. La programación para la mayoría de estas plataformas se realiza a bajo nivel (bare-metal programming) y precisa de mucho trabajo e investigación para conseguir desarrollos eficientes. A todo esto hay que sumar que las arquitecturas paralelas han experimentado un punto de inflexión relevante que ha hecho que la comunidad de desarrolladores se replantee los modelos de trabajo para poder incluir, ahora, de manera explícita el paralelismo y la concurrencia (y cómo hacerlos trabajar juntos) en las nuevas arquitecturas multicore.

El propósito de las abstracciones que se emplean en este trabajo es el de reducir el esfuerzo para poder llevar una aplicación científica a cualquier plataforma con un coste mínimo sin perjudicar el rendimiento, comprobando en qué situaciones éste incluso se mejora.

0.2. Aspectos fundamentales del rendimiento. Aplicaciones científicas

El desarrollo de una aplicación científica requiere en muchas ocasiones el tener que aceptar la construcción de un modelo que simplifique o no contemple todas las posibilidades del fenómeno científico. Es imposible poder considerar todos y cada uno de los eventos y aspectos involucrados en un fenómeno interesante para el estudio. El primer paso del desarrollo, por tanto, es modelar el problema, aceptando determinadas simplificaciones que hagan factible su solución. El segundo

paso es la discretización del modelo para poder transportarlo a un entorno computacional. En este entorno la elección de las estructuras de datos y la forma de operar con ellas o de interaccionar con ellas, y cómo usarlas son el primer problema a resolver. La división del problema en algo susceptible de ser paralelizado es el segundo problema a resolver (supeditado al primero). Resulta que para que una aplicación científica entre en producción, antes han pasado años de desarrollo, pruebas, refinamiento, etc. En esta situación, se suele hacer poco viable que las aplicaciones que hoy día se usan, no se desee que vuelvan a ser rescritas. Por tanto el responsable de paralelizar las aplicaciones se encuentra con dos problemas :

- Asumir las aplicaciones ya desarrolladas y en producción, aunque estén desarrolladas en entornos poco apropiados para las nuevas arquitecturas paralelas.
- Plantear el desarrollo desde cero, en entornos y frameworks más ventajosos, de las aplicaciones científicas venideras.

Frente a estas dos situaciones, ésta puede ser una opción a considerar, capaz de ofrecer un rendimiento aceptable. El rendimiento es el resultado de lo bien que se consigue encajar las características de la aplicación con las características de la arquitectura. Existen diversos medios para alcanzar el rendimiento máximo (explotar al máximo el ILP que consiste en aprovechar el paralelismo probable que existe entre las instrucciones del programa -aunque desde la perspectiva del modelo memoria no es siempre eficiente [9]-, puede ser explotado a través del compilador pero lo normal es que el soporte al ILP venga desde el sustrato hardware; DLP [10], FMA, etc..) [2], usar instrucciones no-FP porque no afectan al ancho de banda, evitar la divergencia de las hebras computacionales (localidad)...etc.

No existe una estrategia común para todas las aplicaciones y todas las arquitecturas cuando se pretende incrementar el rendimiento. De hecho, cada arquitectura tiene un balance diferente entre tres componentes. Y lo mismo ocurre con cada aplicación. Los tres componentes a considerar son:

- Computación : La computación depende desde el compilador [11] usado hasta las directivas de precompilación incluidas en el código para dirigir la compilación del mismo. Evidentemente también depende de cómo el desarrollador escriba el código, pero en muchos casos el compilador es capaz de resolver deficiencias de programación inducidas por la falta de experiencia.

- Comunicación : La comunicación es la piedra angular de un gran número de aplicaciones paralelas, y de un gran número de aplicaciones científicas paralelas.
- Localidad : La localidad entendida ésta como el adecuado emplazamiento de los datos (y su recorrido, por parte de las instrucciones que los usan) con objeto de evitar un incremento en las comunicaciones, podría ser considerada como un apartado del componente anterior (comunicación), pero de máxima importancia ante determinadas aplicaciones. Podríamos decir, por tanto, que es necesario maximizar la localidad para minimizar la comunicación.

Desde este punto de vista, en el desarrollo de una aplicación paralela científica se debe considerar siempre que la computación es gratis y la comunicación es cara. La forma de poder conciliar en muchos casos estos dos elementos es explotando el concepto de localidad.

Una forma de explotar el paralelismo al máximo en las arquitecturas es que el paralelismo sea inherente en el algoritmo [12] (como es el caso de la Orientación a Objetos). El traslado del algoritmo al modelo computacional requiere que el paralelismo sea expresado en la implementación de alto nivel, evitar en lo posible el tener que fragmentar el código en zonas de envío y de recepción de mensajes, etc. Además el paralelismo ha de ser explícito en el código que se genere (las herramientas de traducción han de ser claras en el uso de paralelismo a la hora de generar el código). De este modo el primer componente citado, la computación, será lo más eficiente posible. En términos de comunicación, debemos tratar de consumir el mínimo ancho de banda para poder usar cuantos más canales virtuales mejor. En este sentido se prefieren técnicas de prebúsqueda (software especulativo) [13] y por el otro lado, las arquitecturas más usadas han sido las arquitecturas NUMA. El modelo de memoria juega, en aplicaciones paralelas, un papel destacado y su relación con los modelos de programación es muy estrecha [14].Tradicionalmente el modelo de programación asociado a las arquitecturas NUMA ha sido el paso de mensajes y el asociado a arquitecturas UMA, la programación multihebrada (base del software especulativo). La Orientación a Objetos ([15]), sin embargo, es un modelo apoyado en el envío de mensajes entre objetos lo que la hace útil para las arquitecturas NUMA, y además cumpliendo el requisito impuesto de que en el código generado se encuentra de manera explícita el paralelismo [12], pues en el uso de la orientación a objetos sobre arquitecturas NUMA el paso de mensajes entre diferentes procesadores (además de ser transparente para el programador) es explícito en el uso del código generado en la forma de proxies, tuberías, etc. El modelo de objetos es inherentemente paralelo [15] como se ha citado antes, pero además es capaz de ejecutarse en arquitecturas UMA

sin necesidad de alterar su código, pues el envío de mensajes en este caso se hace convivir con la capacidad concurrente de los objetos [15]. De este modo, el modelo de objetos es capaz de explotar la computación y la comunicación (ésta última a través de la localidad, proporcionada por una interesante característica propia de la orientación a objetos como es la encapsulación, tanto de datos como de código [3]).

0.3. Objetivos

El proceso de reconstrucción 3D de imágenes tomográficas da como resultado básicamente la creación y visualización de estructuras internas, por ejemplo celulares, utilizando técnicas de microscopía electrónica. La reconstrucción de los datos obtenidos mediante estas técnicas es un proceso costoso en tiempo. El problema de reconstrucción apuntado es un proceso de alto coste computacional si queremos obtener imágenes con una precisión adecuada. En nuestro grupo contamos con desarrollos donde se consiguen realizar reconstrucciones de alta calidad, en tiempos de ejecución mínimos. El esfuerzo necesario para mantener estas cotas de calidad en las nuevas arquitecturas, es muy elevado si siguiéramos los procedimientos tradicionales en lugar de abordarlo desde la perspectiva de las abstracciones, comentadas en el punto anterior, dado que los desarrollos se abordan empleando C y MPI lo que infiere poco margen para poder expresar el paralelismo en el código. Además este paradigma de trabajo no es de fácil implementación, no es un paradigma de trabajo donde el paralelismo se puede incorporar de manera sencilla en los desarrollos.

En las aplicaciones de reconstrucción, el componente de computación está muy bien estudiado, pues los algoritmos, los desarrollos paralelos y los métodos empleados son de probada eficacia en este campo de trabajo. El paralelismo se aborda haciendo una división del problema en procesos. Las estructuras de datos empleadas suelen ser las tradicionales, donde lo aconsejable es utilizar estructuras de datos avanzadas donde el paralelismo vaya expresado inherentemente en la definición, avances recientes en este campo pueden encontrarse sobre todo en el uso de plantillas (templates) especiales en lugar de usar estructuras de datos convencionales [7]. El componente de comunicación está inteligentemente minorado con estrategias de envíos y recibos no bloqueantes para poder computar mientras se comunica, pero el cambiar el tamaño del problema puede afectar a las cotas de rendimiento a consecuencia de no poder incluir en el código la posibilidad de ocultar la latencia en las comunicaciones de manera adaptativa. Uno de

los aspectos en los que se centra esta tesis versa sobre cómo explotar la condición de localidad (tanto de datos como de código) de los objetos para habilitar la concurrencia eficientemente y poder solapar las comunicaciones con computación. Evidentemente siempre existe un punto en el que si crece, la comunicación, sería imposible seguir ocultándola. Nuestro objetivo es reducir el impacto en esta situación.

Proporcionar técnicas de alto nivel (tanto para aplicaciones ya desarrolladas, como para aplicaciones en vías de desarrollo) es el primer objeto de trabajo en esta tesis. El siguiente objetivo consiste en proporcionar estrategias robustas de balanceo que ante la aparición de otros usuarios en la máquina paralela o ante situaciones de heterogeneidad en el sistema, el rendimiento de la aplicación se vea perjudicado lo mínimo posible.

De manera agrupada, los objetivos que se pretenden cubrir con este trabajo se pueden resumir en :

- Facilitar la concurrencia. El modelo de computación Orientado a Objetos de por sí no es concurrente, pero existen extensiones que integradas en el modelo lo convierten en una herramienta potente para poder llevar a cabo en un procesador (núcleo) con éxito la ejecución de numerosos flujos de control (configurados como hebras de usuarios) teniendo como consecuencia directa la ocultación de la latencia.

- Explotar la localidad de los datos. En este trabajo se han implementado técnicas para respetar la localidad de los datos en la computación incluso cuando se hace preciso mover carga entre procesadores de modo que la aplicación se *mapee* en la mejor combinación de procesadores de la arquitectura paralela que la ejecuta. Aprovechar al máximo la capacidad de la plataforma heterogénea con estrategias de balanceo dinámico de carga.

- Establecer procedimientos de visualización que faculten el estudio y el análisis de posibles desequilibrios en la carga computacional o situaciones de pérdida de rendimiento en la aplicación.

- Aplicación de todas las técnicas anteriores al problema de la reconstrucción 3D de imágenes tomográficas.

0.4. Estructura de la tesis

Durante el trabajo de esta tesis se ha pretendido hacer frente a los objetivos planteados, capítulo a capítulo. En el capítulo uno se plantea el marco teórico fundamental y el estado del arte. En el capítulo dos se expone el dominio en el que se centra nuestro trabajo, la reconstrucción 3D de imágenes tomográficas. Durante el capítulo tres se exponen las soluciones adoptadas para paralelizar la aplicación del dominio del problema, los resultados de la paralelización se muestran aquí. El capítulo cuatro versa esencialmente sobre las técnicas de balanceo de carga para optimizar la asociación entre la aplicación y la arquitectura paralela (computación, comunicación y localidad). El capítulo cinco queda dedicado a resaltar las conclusiones y a apuntar el trabajo futuro.

0.4.1. Capítulo 1: objetivos

Existen problemas que por su complejidad son difícilmente resolubles en un periodo de tiempo razonable utilizando procedimientos basados en la ejecución secuencial de un programa. Sin embargo existen técnicas que consisten en el uso de un conjunto coordinado y cohesionado de procesadores que han de actuar como uno sólo en la dirección de la solución al problema complejo planteado. Estos procesadores actúan como un gran computador en el que cada componente de procesamiento trabaja en paralelo. Existen diferentes arquitecturas de agrupación de procesadores. Cada arquitectura paralela es útil para un tipo de problema diferente.

El procesamiento paralelo es por tanto una alternativa al procesamiento secuencial. En el procesamiento paralelo cada elemento de procesamiento que compone la arquitectura en la que se aplica la solución, trabaja en paralelo con los demás, reduciendo el tiempo que toma la resolución del problema pues pueden llevarse a cabo varias operaciones de manera simultánea.

El extendido uso de estas arquitecturas está ampliamente justificado en la investigación, tal es así que parte importante de la investigación relacionada con la resolución de problemas complejos se centra en los *algoritmos paralelos* y en las *metodologías y modelos de programación paralela.*

Las técnicas de implementación que han de portar el problema a estas arquitecturas son diversas y así como cada arquitectura posee peculiaridades, igualmente son particulares. La combinación de arquitectura e implementación de la solución al problema constituyen un binomio esencial desde la perspectiva del problema.

En este capítulo se presenta la guía argumental del trabajo desarrollado en este trabajo de tesis.

0.4.2. Capítulo 2: objetivos

Este capítulo presenta la problemática de la reconstrucción tomográfica en biología estructural desde la perspectiva de la computación de altas prestaciones. La metodología de reconstrucción 3D se caracteriza por un gran coste computacional y el empleo de computación paralela resulta esencial para su puesta en práctica. En este capítulo se muestran de forma general las aproximaciones paralelas que hasta el momento se han realizado. También se exponen las estrategias sobre las que se está trabajando en la actualidad, centradas principalmente en computación distribuida sobre sistemas heterogéneos.

0.4.3. Capítulo 3: objetivos

La computación, en general, siempre ha estado supeditada al procesador, en especial la computación de altas prestaciones y la computación paralela. Obtener rendimiento en los desarrollos ha sido tradicionalmente complejo, por lo que siempre se ha tendido a elegir modelos de programación que no añadan más complejidad que la estrictamente necesaria. En [16] se advierte sobre la relación entre complejidad del procesador y ganancia en rendimiento de este. La relación no es esperanzadora, esto unido al elevado y cada vez mayor consumo energético de los procesadores (*powerwall* [17]), y los problemas de disipación térmica [18], entre otros, han forzado a adoptar cambios que,hasta ahora, la industria había rechazado [19].

Los procesadores multicore y las plataformas de computación donde se combinan GPUs y CPUs, se están convirtiendo, cada vez más, en atractivas plataformas para los desarrolladores de aplicaciones eminentemente científicas o de simulación. Estas arquitecturas se caracterizan por ser transgresoras y constituir un punto y aparte en la tendencia de la computación de altas prestaciones. Los procesadores multicore son entidades computacionales construidas a partir de un agregado de unidades de procesamiento pseudo independientes denominadas *cores o núcleos*. La tendencia en la computación paralela se está desplazando hacia el área de los procesadores multicore. En este escenario aparece un factor con el que las aplicaciones tradicionales paralelas deben enfrentarse, *la concurrencia*. En este trabajo, se reduce el pesimismo que en relación a las ganancias en rendimiento de estas nuevas arquitecturas, *M. Hill y M. R. Marty* aventuraban en

[20]. Todo consiste en adoptar un giro en el modelo de computación, tal y como indican muchos autores [21],[22].

0.4.4. Capítulo 4: objetivos

La mayoría de las aplicaciones científicas se caracterizan por hacer un uso intensivo de los recursos computacionales. En base a esto se ha tendido a emplear plataformas de computación paralela como clusters porque son el mejor medio para acometer tareas cuyo rendimiento en un sólo procesador no es asumible. Además son una plataforma relativamente barata, si las comparamos con los grandes sistemas multicomputadores. De ahí el gran auge que la plataforma cluster ha experimentado durante la última década. No obstante, el rendimiento de las aplicaciones ejecutadas en clusters puede verse seriamente afectado por factores externos a la propia aplicación. Entre estos factores se puede mencionar al uso compartido de los procesadores del cluster, las latencias desiguales en tramos de la red de conexión, la arquitectura de cada procesador parte del cluster, etc.... Todo esto puede llegar a ocasionar que la ejecución no progrese por igual en todos los procesadores.

Incluir hebras de usuario en el desarrollo de una aplicación hace más sencillo aplicar técnicas de balanceo de carga (migrar una hebra de usuario de un procesador a otro es menos costoso que migrar un proceso [23], [24]), de este modo se puede hacer que la aplicación se adapte al entorno de ejecución para reducir el impacto de los factores externos.

Independientemente de si se usa un *runtime* o la política de balanceo de carga se *embute* en la misma aplicación, lo realmente importante es la selección de la estrategia de balanceo de carga apropiada. La política que se aplicará para mantener la computación balanceada no sólo depende de la aplicación también está muy ligada al tipo de arquitectura hardware sobre la que se ejecuta.

En este capítulo se describe y se analizan dos estrategias de balanceo **dinámico** de carga. Estas estrategias se han diseñado para las plataformas cluster y multicore. Las estrategias de balanceo que se describen en este capítulo se basan en hebras (se aplican a las aplicaciones desarrolladas en el capítulo 3 de esta tesis). Una de las principales características de las políticas de balanceo de carga presentadas es la preservación de la localidad de datos durante las operaciones de migración. La implementación de estas dos estrategias se apoyan en AMPI [25] (capa de abstracción del framework Charm [26]).

La posibilidad de observar, sin interferir en el progreso de la aplicación, las condiciones de ejecución de la misma hacen necesario el desarrollo de un entorno de visualización con el que se pueda observar la eficacia de las estrategias de balanceo implementadas. Este capítulo termina mostrando una herramienta sencilla de visualización. La presentación visual del progreso de la aplicación puede llegar a ser un método efectivo con el que identificar aspectos sutiles que de otro modo quedarían ocultos. La práctica de usar visualizadores gráficos está muy extendida en el entorno de las aplicaciones científicas paralelas tales como el desarrollado por *B. Gregg* [27].

0.4.5. Capítulo 5: conclusiones

El trabajo desarrollado en esta tesis ha consistido proponer nuevos modelos de desarrollo para aplicaciones de elevado coste computacional. Las propuestas se han estudiado sobre aplicaciones del ámbito de la microscopía electrónica de alta resolución, concretamente en aplicaciones diseñadas para reconstruir volúmenes a partir de un conjunto de proyecciones obtenidas desde estos microscopios. Durante todo el trabajo ha estado presente el *aforismo de Nickalus Wirth* donde se considera que un programa no sólo son sus instrucciones sino también los datos sobre los que se opera, de este modo portar aplicaciones como las que se presentan en este campo de trabajo supone no sólo optimizar el algoritmo sino además prestar atención a cómo éste trabaja con los datos asociados. Desde este punto de vista se ha trabajado con el modelo orientado a objetos. La caracteristica secuencial presente en los algoritmos empleados y las arquitecturas sobre las que se ha trabajado invitan a pensar en la programación multihebrada como solución a problemas como el de la ocultación de la latencia. Una de las conclusiones más importantes que se desprende de este trabajo es que se pueden combinar multihebra y orientación a objetos para generar un nuevo espacio de trabajo donde las latencias se puedan ocultar de manera sencilla.

Otro de los problemas abordados en este trabajo ha sido el de la adaptabilidad. Las aplicaciones tradicionales son poco flexibles y una mala distribución inicial del trabajo puede lastrar bastante el resultado de la computación. El modelo de trabajo que se ha adoptado facilita la migración de trabajo entre procesadores y por ende la capacidad de adaptar la computación al entorno sobre el que se lleva a cabo. Estudiar este aspecto y proponer estrategias de balanceo para aplicaciones de reconstrucción de imágenes es uno de los objetivos de este trabajo.

El estudio de estas aplicaciones, la optimización y monitorización de las aplicaciones (para las que hemos creado una aplicación de visualización) hacen que existan numerosas posibilidades

para continuar trabajando. Además de estudiar aspectos tan interesantes como el de poder portar la aplicación de reconstrucción a arquitecturas que puedan aparecer en el futuro, lo que hace necesario desarrollar capas de abstracción donde poder ejecutar las aplicaciones.

Capítulo 1

Marco Teórico Fundamental

1.1. Computación de altas prestaciones

Existen problemas que por su complejidad son difícilmente resolubles en un periodo de tiempo razonable utilizando procedimientos basados en la ejecución secuencial de un programa. Sin embargo existen técnicas que consisten en el uso de un conjunto coordinado y cohesionado de procesadores que han de actuar como uno sólo en la dirección de la solución al problema complejo planteado. Estos procesadores actúan como un gran computador en el que cada componente de procesamiento trabaja en paralelo. Existen diferentes arquitecturas de agrupación de procesadores. Cada arquitectura paralela es útil para un tipo de problema diferente. El procesamiento paralelo es por tanto una alternativa al procesamiento secuencial. En el procesamiento paralelo, cada elemento de procesamiento que compone la arquitectura en la que se aplica la solución, trabaja en paralelo con los demás, reduciendo el tiempo que toma la resolución del problema pues pueden llevarse a cabo varias operaciones de manera simultánea. El uso extendido de estas arquitecturas está ampliamente justificado en la investigación, tal es así que parte importante de la investigación relacionada con la resolución de problemas complejos se centra en los *algoritmos paralelos* y en las *metodologías y modelos de programación paralela.*

Las técnicas de implementación que han de portar el problema a estas arquitecturas son diversas y así como cada arquitectura posee peculiaridades, igualmente son particulares. La combinación de arquitectura e implementación de la solución al problema constituyen un binomio esencial desde la perspectiva del problema [9], [14] e incluso se puede decir que dada la complejidad de la programación en las últimas arquitecturas disponibles, se recomienda adoptar

fuertes procesos de ingeniería del software, los trabajos [28] y [29], son un ejemplo. La unión de estas arquitecturas y de sus modelos específicos de programación se ha denominado *computación de altas prestaciones*. La definición del término *Computación de Altas Prestaciones* según el RFC-1983 *Internet User's Glossary - Network Working Group*, se puede entender como:

> ***High performance computing encompasses advanced computing, communications, and information technologies, including scientific workstations, supercomputer systems, high speed networks, special purpose and experimental systems, the new generation of large scale parallel systems, and application and systems software with all components well integrated and linked over a high speed network.***

De la definición previa, se debe subrayar el término *large scale parallel systems* o *sistemas paralelos de gran escala* como arquitectura que define a la *amalgama de procesadores* que forman a la máquina paralela. Como consecuencia del uso de estas arquitecturas, las comunicaciones y el software han de ser específicos para la misma y para el problema.

Las técnicas de computación de altas prestaciones, incluido el paralelismo, intentan dar respuesta a numerosos problemas de Ciencia e Ingeniería, que requieren el procesamiento de grandes cantidades de datos numéricos [30], reduciendo los tiempos de resolución, y posibilitando el tratamiento de problemas de mayor complejidad. Algunos pocos ejemplos son el análisis de la variabilidad climática [31] [32], el cálculo del campo gravitatorio terrestre [33], el diseño de circuitos electrónicos VLSI [34], o la detección de partículas cósmicas [35], entre otros.

Por lo tanto, dentro del ámbito de la *computación de altas prestaciones* en el que se centra el trabajo de esta tesis caben citar tres aspectos:

- Arquitecturas paralelas.
- Metodologías software y su relación con la arquitectura paralela.
- Redes de comunicaciones de alta velocidad.

1.1.1. Estado actual

El aumento en el rendimiento de los procesadores durante la última década ha avanzado, aparentemente, de manera indefinida (en [36], concretamente en el capítulo 25 se detalla el avance

del rendimiento en las arquitecturas paralelas). El límite de este aumento en el rendimiento viene impuesto por las leyes de la física. Hasta ahora las vetas de rendimiento más aprovechadas eran: *la escala de integración*(incrementar el numero de puertas por unidad de superficie), *aumentar la frecuencia del reloj*, *y el consabido Instruction Level Parallelism o ILP*. Al principio del año 2003 este crecimiento al que nos tenía acostumbrado la industria con las tecnologías de semiconductores utilizadas se acercó al límite impuesto por la física (consumo y disipación del calor) lo que impide que se siga incrementando la frecuencia del reloj. Si no se puede seguir incrementando la velocidad de reloj, se pretendió que la responsabilidad recayera sobre el ILP que comenzó a evolucionar para aplicar técnicas muy complejas de paralelismo (predicción de salto, ejecución especulativa, etc) que requerían superficies prohibitivas dentro del chip, que no eran del todo eficientes [9] (y con algo más de detalle, se puede consultar los problemas del ILP en los capitulos 2 y 3 en [37]). Hoy por hoy, la única veta por explotar para alcanzar los incrementos de rendimiento a los que el software está acostumbrado es el número de puertas por unidad de superficie[38] (en este sentido es interesante revisar la entrevista realizada a Charles P. Thacker [39]). De acuerdo a las expectativas, el nivel de integración se duplicará cada año. Tan ingente cantidad de puertas invitan a crear más núcleos dentro del mismo procesador, esperando que en 2012 se excedan los 32 núcleos. El inconveniente es que sólo se apreciará el impacto de esta técnica en el software, en términos de rendimiento, si éste es capaz de escalar a través de todos estos núcleos [40], [41]. Con la aparición de las CPUs multicore y del inicio de la computación en arquitecturas manycore se puede considerar que un chip individual, ahora, es un sistema paralelo per-se. Es más, con esta tecnología, y salvadas las limitaciones de las que acabamos de hablar, el paralelismo de estas nuevas arquitecturas escala de acuerdo a las **leyes de Moore**[42, 43, 44, 45]. El verdadero reto se ha trasladado al desarrollo de aplicaciones científicas que escalen, de manera transparente, a la par que se añaden nuevos núcleos a la arquitectura.

La evolución de las arquitecturas paralelas (hardware en el que se apoya la computación de altas prestaciones) sigue acuñando a los sistemas con más de un núcleo como arquitecturas que sustituirán a las arquitecturas paralelas convencionales (clusters). Hay mucho software desarrollado para las arquitecturas que hasta hoy son las arquitecturas paralelas por excelencia (clusters) pero la aparición de los multicore hace necesario que se porten todas las aplicaciones y que se adopten nuevas estrategias de desarrollo (metodologías de software apropiadas [46]) que

permita disfrutar del escalado trasparente y de las consecuencias directas en el rendimiento [41]. Es importante notar que la tendencia apunta a potenciar el escalado de las aplicaciones antes que el máximo aprovechamiento de los recursos que se les ofrece por core (una de las últimas tendencias en las arquitecturas de procesadores en esta línea consiste en eliminar las cachés y aprovechar ese espacio para incluir circuitería [47] que pueda hacer otras *tareas* mientras se resuelve el fallo de página).

1.1.2. Computación paralela

El modelo *uni-hebrado* de Von Neumann es comprensible porque es determinista. La programación paralela es bastante compleja [29]. Además, el código paralelo está sujeto a errores (deadlocks, livelocks, condiciones de carrera, etc) que pueden ser extremadamente sutiles y de muy difícil detección (muestra de esto se puede encontrar en [48], concretamente en el el capitulo 10) a menudo porque el error en cuestión no sigue un patrón de ejecución determinista [37] (capitulo 4), bien es cierto que cada vez más se adoptan técnicas que permiten acotar errores y facilitar la programación paralela, como ejemplo de esto sirva el trabajo de *Kelly, T. et al* [49] o los trabajos [50] y [51]. Uno de los grandes cambios conceptuales entre la programación secuencial y la paralela es que en la secuencial el uso de abstracciones guia el desarrollo por ejemplo, el uso de las sentencias como *goto* ha sido desterrado de la mayoría de los lenguajes (como poco ha sido inteligentemente ocultado con complejas estructuras de control) sin embargo, con la programación paralela es como tenerlos repartidos por todo el código durante la ejecución del programa. No obstante, esta tendencia está cambiando ya que cada vez más es común ver cómo los lenguajes y herramientas, empleados para la programación paralela, ofrecen el uso de abstracciones que por un lado ayudan a generar desarrollos robustos y por el otro ayudan a extraer mejor el paralelismo [28]. Un desarrollo paralelo determinista es todo un reto (para una revisión del sistema de planificación en sistemas paralelos, pueden consultarse los capítulos 20 a 23 de [52]). El procesamiento paralelo ha causado un gran impacto en muchas áreas científicas ([48, 52])donde se precisa, dada la naturaleza de los problemas que pretenden resolver, una alta capacidad de cómputo [46]. Ejemplos de esto, puede encontrarse en aplicaciones como la visualización de imágenes médicas [53], simulaciones [54], predicciones científicas, etc. . . , en [48] concretamente en el capitulo 29 de la tercera parte del volumen se explica cómo abordar y modelar el desarrollo paralelo de aplicaciones biomédicas.

Un reto tan sutil como el anterior, es el de conseguir que el rendimiento de una aplicación paralela escale. De acuerdo a la **ley de Amdahl** [55, 56], el límite teórico del speedup debido a la paralelización de un programa depende de la proporción de código que no es paralelizable (si el 10 por ciento de un código a paralelizar, es inexorablemente secuencial, entonces aun ejecutándose en un número infinito de procesadores, jamás alcanzará un speedup igual o superior al secuencial por diez). Esta secuencialidad que lastra el código paralelo es inevitable ya que todo algoritmo susceptible de ser paralelizado impone cierto carácter secuencial a fin de conservar el orden de las operaciones.

Tradicionalmente se ha distinguido entre dos tipos diferenciados de paralelismo, el paralelismo a nivel de instrucción o ILP (en ocasiones ineficiente y complejo [9]) y el paralelismo de tareas. Siendo el primero más que un paralelismo explícito un aspecto arquitectónico del procesador que permite la ejecución simultánea de instrucciones en el mismo. Y el segundo, un aspecto de las aplicaciones que se ha de explotar explícitamente y en función de la arquitectura subyacente. En cualquier caso, en ambos niveles de *paralelismo* se sufren la secuenciación, tan necesaria para mantener el orden lógico de las operaciones (se ha estudiado extensamente cómo limitar el efecto de la porción de código secuencial, una de las más útiles soluciones aportadas es la del uso de una *abstracción* denominada *futures*, esta abstracción permite la ejecución de operaciones asíncronas remotas mientras continua la ejecución local en paralelo, incluso cuando hay dependencias entre ambas ejecuciones [57], autores como Goldstein [58] muestran como este modelo permite relajar estas condiciones de secuencialidad), todo esto, por supuesto, siempre al amparo de los modelos de memoria que tienen la última palabra [14]. En lo que a abstracciones respecta, existen otros modos de paralelismo que usan abstracciones para extraer el paralelismo de los programas (no es un paralelismo al nivel de ILP o TLP, donde existe soporte hardware y del lenguaje, es un paralelismo a más alto nivel), un ejemplo de este tipo de paralelismos es el *paralelismo basado en pipeline*, este paralelismo -basado en abstracciones- está comenzando a hacerse notar gracias a las nuevas arquitecturas de procesadores y en los desarrollos paralelos recientes tal y como se puede ver en los trabajos [46], [49], [28], [29], [50] y [51],.

Aunque la ley de Amdhal [59] es una guía muy útil e intuitiva para las expectativas de una paralelización, lo cierto es que llegar a determinar, con certeza, la proporción de código que realmente se ejecutará en paralelo es una tarea imposible. La serialización del código puede ocurrir en cualquier momento como consecuencia de la resolución de un conflicto por acceso

compartido a un recurso o por tratar de acceder a localizaciones de memoria demasiado alejadas. Si es posible minimizar su efecto tal y como postula la ley de Gustafson [60] considerando la carga (W) del sistema, escalable.

Los métodos tradicionales usados en la programación paralela (paso de mensajes, uso de monitores si se usan hebras, ...) a menudo sufren de trabas en la escalabilidad ya que estos mecanismos pueden precisar etapas de serialización que de hecho se incrementan al incrementar el número de núcleos. Si cada núcleo tiene que sincronizarse con un sólo núcleo esto haría que la serialización creciera linealmente, si en lugar de esto la sincronización ha de realizarse entre todos los núcleos hábiles, entonces estaríamos afrontando un incremento combinatorio en la serialización. En definitiva, cualquier código que sea serializado no alcanza el rendimiento que se podría esperar si tuviéramos únicamente en cuenta el número de núcleos usado.

Por todas estas razones, trasladar una aplicación al paradigma paralelo es una actividad tediosa y laboriosa. Someter todos los efectos y sutileza de la naturaleza no-determinista de la ejecución de códigos paralelos hasta llegar a un nivel aceptable puede llevar años. Si tenemos en consideración a los procesadores multicore, es probable que cuando el software haya alcanzado con limpieza el nivel de escalabilidad adecuado, la arquitectura subyacente ya no sea la más adecuada ya que excederá el nivel de paralelismo para el que la aplicación, ahora, está preparada. Desgraciadamente, según apuntan las previsiones, el nivel de paralelismo crecerá a un ritmo mucho más elevado del que nuestro software tal como hoy lo concebimos puede escalarlo. Se ha invertido mucho esfuerzo en hacer practicable la paralelización, muestra de ello es la existencia de grupos de investigación (*Parallel Programming Models and Compilers*, Universidad de Málaga) y proyectos consolidados (como shape analyzer [61], desarrollo de este grupo) que trabajan en la paralelización automática de código. Esta técnica consiste en lograr que el compilador sea capaz de extraer (en la mayoría de las ocasiones a través de análisis estáticos) características de la aplicación para poder determinar una paralelización eficiente y desatendida. A pesar de que existen técnicas muy depuradas, no en todos los casos se llega a alcanzar una paralelización óptima.

Estos problemas mencionados justifican la búsqueda de un nuevo paradigma, que permita que la aplicación se adapte a la nueva arquitectura alcanzando unos niveles aceptables de eficiencia y facilitando posteriormente la adaptación idónea, en manos del desarrollador. El modelo buscado no pretende que dada una arquitectura la aplicación obtenga un escalado cercano al idóneo sino

que permita que la aplicación escale bien (por ejemplo, con un número de núcleos variable y en ascenso) sin necesidad de aplicar cambios al código. En parte esto es posible si se logra un nivel de granularidad mucho más fino para el paralelismo.

1.1.2.1. Instruction Level Parallelism (ILP)

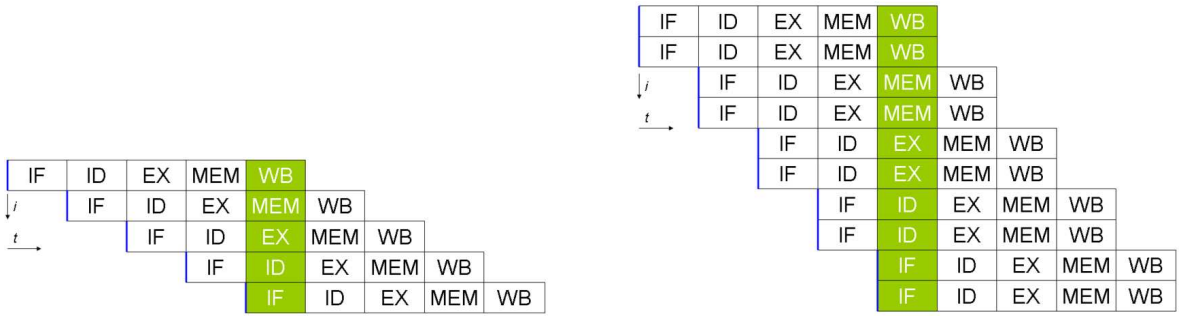
La búsqueda de la *ejecución simultánea* es algo que viene motivando a la comunidad científica desde los inicios de la computación (en [62], concretamente en los capítulos 4 y 6 se puede encontrar una revisión detallada sobre el ILP y VLIW, ambos capítulos muestran el interés y el esfuerzo de la comunidad científica por conseguir la citada ejecución simultánea). El *Bit Level Paralelism o Paralelismo de Bits* fue el precursor de todos los esfuerzos empleados en conseguir que el procesador trabaje con un grano de paralelismo cada vez más fino. Desde el momento en que la escala de integración *VLSI* llegó a implantarse de manera definitiva en los chips destinados a los computadores, el *Speed-Up* se mantenía duplicando el tamaño de la palabra, de esta forma se conseguía reducir el número de instrucciones ejecutadas por el procesador (por ejemplo, un procesador de 8 bits precisa usar dos instrucciones de suma para sumar dos enteros de 16 bits en contraste con los procesadores de 16 bits que sólo requieren una instrucción).

El ILP es una magnitud que informa de la cantidad de operaciones, correspondientes a un programa, que pueden realizarse simultáneamente. Un programa es, a fin de cuentas, un conjunto de instrucciones que han de ser ejecutadas por el procesador. Estas instrucciones pueden reordenarse y ser agrupadas para ejecutarlas en paralelo sin que el resultado final se vea alterado. El ILP se beneficia de dos aspectos arquitectónicos del procesador muy importantes:

- Los cauces de instrucciones.
- La replicación de las unidades funcionales críticas.

El cauce clásico posee cinco etapas como refleja la fig.1.1. En la fig. 1.1(a) se puede observar que un procesador con este cauce puede tener en ejecución casi simultánea a cinco instrucciones. Estas etapas pueden *paralelizarse*. La paralelización en este caso viene a través de la inclusión de unidades funcionales duplicadas que permiten ejecutar dos instrucciones del mismo tipo en *paralelo* como puede apreciarse en la fig. 1.1(b).

No obstante la dependencia que existe entre las instrucciones es un aspecto delicado a la hora de extraer rendimiento de los cauces en los procesadores. Considérese la figura 1.2. Estas instrucciones pueden ejecutarse, por ejemplo en cualquier procesador MIPS (arquitectura RISC).



(a) Cauce clásico de cinco etapas

(b) Cauce súper escalar de cinco etapas

Figura 1.1: Representación de los cauces clásicos de instrucciones como medios de implementación del ILP

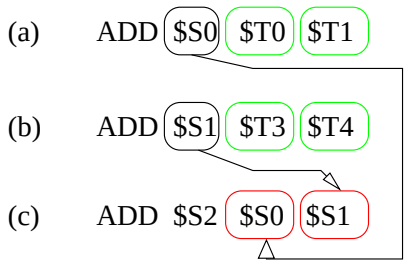


Figura 1.2: Secuencialización para mantener el orden.

En esta figura (fig.1.2) se puede observar cómo en un cauce como el representado en la fig.1.1(a), la ejecución de la instrucción (b) ha de esperar a que (a) termine de usar la unidad de suma. La instrucción (c) así mismo tendrá que esperar a que termine la instrucción (b) debido a la dependencia que fig.1.2 manifiesta. Si se usara en su lugar un cauce súper escalar para acelerar la ejecución, sólo las instrucciones etiquetadas (a) y (b) podrían ser emitidas para su ejecución simultánea en unidades funcionales separadas (no poseen dependencias de datos). Como se puede observar, la instrucción etiquetada como (c) no es posible ejecutarla en *paralelo* pues es preciso que ésta espere a que las dos instrucciones previas terminen. Si asumiéramos que cada instrucción puede terminar su trabajo en una unidad de tiempo, podríamos afirmar que el *ILP es 3/2*. En este caso, el cauce se degrada e indirectamente la ganancia (*Speed-Up*) potencial de la computación.

Las técnicas ILP permiten que el procesador y el compilador reorganicen el código para que ejecuten instrucciones de manera solapada fig.1.1. El inconveniente que presentan las técnicas ILP es que dependen en exceso de la naturaleza del problema (las aplicaciones de computación gráfica y científica son aplicaciones en las que el ILP puede llegar a ser extraordinariamente

elevado). Además, la técnica ILP es muy poco efectiva ocultando latencias de memoria (memory stalls) [9] haciendo que el sistema de memoria sea un cuello de botella muy peligroso para la escalabilidad y el rendimiento. Este problema, latencia de memoria, aparece porque los procesadores basados en ILP (capaces de ejecutar instrucciones fuera del orden del programa) son incapaces de realizar el solape entre instrucciones fallidas de lectura, que son las que motivan el problema. En cierto modo, este problema es extensible a las arquitecturas paralelas, donde ocurre lo mismo (ocultación de latencia) pero en este caso con la red.

Las técnicas que se suelen emplear para extraer el máximo rendimiento del ILP son :

- Encauzamiento de instrucciones, donde la ejecución de varias instrucciones puede solaparse.
- Ejecución superescalar ([63]), técnica que consiste en replicar las unidades funcionales de ejecución. Las instrucciones que se ejecutan en paralelo suelen ser las adyacentes en el código. Esta característica suele depender de la primera.
- Ejecución desordenada. Se usa para evitar, en la medida de lo posible, que dos instrucciones con dependencias de datos ((a) y (c)) estén tan próximas que (c) pueda llegar a iniciar su ejecución sin que (a) haya proporcionado el resultado.
- Renombrado de registros.Se usa para evitar las serializaciones innecesarias del programa. Se emplea para permitir la ejecución desordenada.
- Ejecución especulativa y predicción de saltos. Se usan para evitar que el procesador tenga que retirar instrucciones que ya entraron en el cauce.

Dada la dificultad que plantean las técnicas de ejecución desordenada para escalar, últimamente se han revisado las arquitecturas en las que en una misma instrucción se especifican, *explícitamente*, varias operaciones independientes. Las arquitecturas que cumplen tal especificación son las *VLIW* (Very Long Instrution Word) como es el caso del procesador de Intel Itanium IA-64 y las *EPIC* (Explicitly Parallel Instruction Computing) cuyo ejemplo podemos encontrar en los procesadores Itanium e Itanium-2.

El ILP ha sido una técnica ampliamente usada los últimos años para obtener mejoras en el rendimiento de las aplicaciones (sin necesidad de reescribirlas) en contraste con las diferencias, cada vez mayores, entre las elevadas frecuencias de trabajo de los procesadores y los tiempos de

acceso a memoria. En las arquitecturas actuales, un fallo de caché puede derivar en un acceso a memoria principal de hasta varias centenas de ciclos de CPU. Es cierto que es posible tolerar tales latencias de memoria con técnicas ILP, no obstante los costes asociados a tal esfuerzo (consumo y disipación de calor) son prohibitivos, incluso es posible obtener como resultado la reducción de la frecuencia de trabajo del procesador, decreciendo así la ganancia en rendimiento. Por tanto las técnicas ILP muestran ser inadecuadas para la difícil labor de evitar que la CPU quede ociosa esperando ciclos y ciclos hasta que se de por terminado el acceso a memoria principal. La consecuencia directa de este problema es la búsqueda de paralelismo a un nivel más elevado. Las técnicas que se están barajando en la actualidad son el multiprocesamiento y la computación multihebrada [41].

1.1.2.2. Task Level Parallelism (TLP)

El TLP (también denominado paralelismo de control) es una técnica de paralelización de código en arquitecturas de computación paralela. Se centra en la distribución de unidades de ejecución (hebras) en un conjunto de nodos diferentes que trabajan en paralelo. Es una técnica de paralelización que contrasta con el *paralelismo de datos* que consiste en la distribución del conjunto de datos, exclusivamente.

En un sistema multiprocesador, el paralelismo a nivel de tarea se alcanza cuando cada procesador ejecuta hebras diferentes (o procesos diferentes) sobre un conjunto de datos ([36] capítulos 27 y 28 y [37] capítulo 4). Trabajen o no sobre el mismo conjunto de datos, las hebras (o procesos) se comunican entre ellas mientras trabajan. La comunicación se lleva a cabo a través de un flujo de comunicaciones controlado y pre-establecido.

Si se pretende ejecutar código en un sistema con dos procesadores (CPUs **A** y **B**) y se pretende realizar dos tareas T_a y T_b, es posible indicarle a la CPU **A** que acometa la ejecución de la tarea T_a y a la CPU **B** que haga lo propio con la tarea T_b, de manera simultánea [64].

El paralelismo de tareas (TLP) enfatiza la naturaleza distribuida (paralela) de la computación (por ejemplo con hebras) en contraste al paralelismo de datos. La mayor parte de las aplicaciones del campo de la ingeniería se hallan entre el paralelismo de tareas y el de datos [30]. La figura 1.3 muestra un pseudocódigo distribuido entre dos CPUs. Sin embargo el paralelismo de datos, cuyo progreso se detuvo en la década pasada, ahora comienza a tomar protagonismo con la aparición de arquitecturas como las GPUs (Graphic Processing Units) [65], [66].

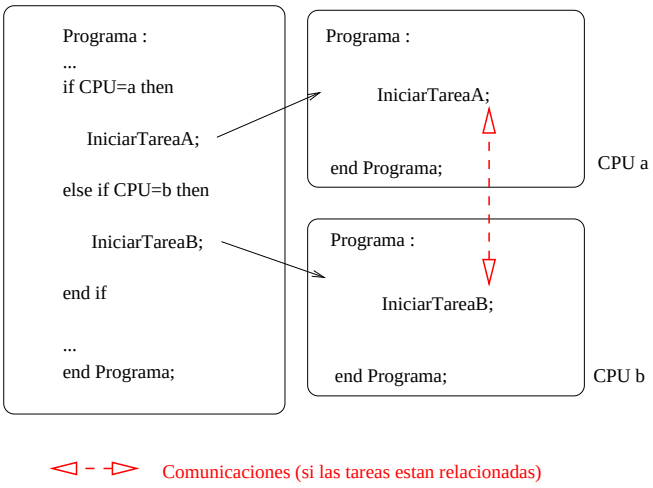


Figura 1.3: Distribución de la tarea neta A+B en dos CPUs.

1.1.2.3. Pipeline Parallelism (PL). Paralelismo basado en Abstracción

Este tipo de paralelismo se apoya en *abstracciones*, constante usada en este trabajo de tesis. Las abstracciones están siendo especialmente de utilidad para expresar tanto los programas ya desarrollados como aquellos por desarrollar en un estilo de paralelismo determinado [28]. El uso de abstracciones es especialmente interesante cuando se pretende una computación dinámica y adaptable, un ejemplo de cómo usar abstracciones para adaptar código ya existente a un entorno mucho más favorable se puede encontrar en [67]. La necesidad de paralelizar código ya existente es evidente, especialmente contando con las arquitecturas multicore. La paralelización puede aplicarse tanto a códigos paralelos como a códigos secuenciales existentes que se deseen paralelizar.

No todos los tipos de paralelización se aplican de igual forma y no existe por tanto una técnica común a todas las paralelizaciones, aunque se han llegado a crear patrones (abstracciones) para facilitar al programador la tarea de la paralelización [51]. No obstante algunos tipos de paralelismos cuentan con un buen soporte en el lenguaje de programación, un ejemplo de esto es la programación multihebrada como soporte al paralelismo de tareas (TLP), OpenMP para el paralelismo de datos, etc. Sin embargo, el paralelismo basado en pipeline no ha sido explotado convenientemente, el Pipeline Paralelism o PL consiste en dividir un bucle en varias etapas y hacer que estas se comuniquen a modo de pipeline [68]

El paralelismo de pipeline constituye una importante abstracción [51] (ya no está apoyada en el soporte hardware como las dos anteriores) útil para programas nuevos como ya existentes y

que todos los desarrolladores deberían poder tener a mano. Este tipo de paralelismo suele estar presente, pero oculto, en todo tipo de códigos secuenciales. En primera instancia, los bucles con dependencias entre iteraciones sólo pueden ser sometidos a esta paralelización usando un mapeo de pipeline paralelo, es evidente que el paralelismo a nivel de datos es imposible. En segundo lugar, el paralelismo basado en pipeline es mucho más eficiente que el paralelismo de datos debido a una mejor localidad tanto en instrucciones como en datos, dentro de cada etapa del pipeline (además en multicores, la comunicación entre núcleos es uno a uno, evitando los scatters y gathers). En relación al paralelismo de tareas, el paralelismo basado en pipeline ofrece una mejora muy atractiva y es que las variables compartidas pueden ser comunicadas en una manera determinista (al estilo productor–consumidor) eliminando las condiciones de carrera en el acceso a los datos [69].

1.1.3. Escalabilidad y rendimiento en aplicaciones de cómputo intensivo

La teoría que sustenta al paralelismo se ampara en un conjunto de leyes fundamentales que impone límites en la ganancia que se espera de la paralelización. Antes de citar las leyes, se debe establecer el objetivo. En general en la computación paralela el objetivo fundamental es bien realizar la mayor cantidad de trabajo en el menor tiempo posible (high-throughput computing) o reducir el tiempo de un procesamiento de un trabajo (high-performance computing). Se puede definir la potencia de un sistema de cómputo como el cociente entre la cantidad de trabajo realizada y el tiempo empleado. La pretensión es optimizar el binomio potencia-coste o coste-beneficio. La relación coste-beneficio derivada del incremento de potencia de un procesador no es lineal y además es muy pobre. Evidentemente la estrategia más acertada para incrementar el rendimiento del sistema computacional, evitando relaciones coste-beneficio tan pobres, consiste en añadir más procesadores a la resolución del problema (motivación que ha hecho de los clusters una arquitectura de moda [70]).

En un escenario ideal, un trabajo computacional susceptible de ser fraccionado en N subtareas donde cada una de esas tareas sea procesada por un procesador diferente, ha de ser capaz de finalizar en $\frac{1}{N}$ respecto al tiempo empleado en finalizar la tarea secuencial. De este modo se espera obtener una ganancia de rendimiento incrementada en un factor de N. Sin embargo en un escenario real, cualquier porción de código paralelizable mantendrá, sin excepción, una porción de código que **debe** ejecutarse secuencialmente. Esta sección de código secuencial no se ejecuta

más rápido en un procesador que forma parte de una arquitectura paralela de lo que lo hacía cuando se ejecutaba en un procesador convencional (incluso puede perder tiempo de ejecución). Sólo la fracción de código paralelizable es capaz de alcanzar una ganancia de N veces respecto a su ejecución secuencial.

El *Speed-Up* de un programa paralelo se define como el ratio entre el tiempo empleado en finalizar la tarea en un procesador (versión secuencial del código) y el tiempo empleado en finalizar la tarea cuando participan N procesadores [56, 71]. Consideremos que *la tarea computacional* es una tarea arbitraria. Se puede definir el *Speed-Up* (incremento en la potencia de cómputo en función de N) en términos del tiempo necesario para completar la ejecución de la tarea en N procesadores.

Sea $T(N)$ el tiempo necesario para terminar la ejecución de un trabajo en N procesadores. Entonces, el *Speed-Up* $S(N)$ se define como el ratio $S(N) = \frac{T(1)}{T(N)}$.

A menudo el tiempo $T(1)$ se compone de una fracción secuencial y otra paralelizable. El tiempo de la fracción secuencial no disminuye cuando se paraleliza el problema. La fracción paralela, en contadas ocasiones alcanza a reducirse en un factor $\frac{1}{N}$. De aquí que una definición algo más formal para el *Speed-Up* sea:

$$S(N) = \frac{T(1)}{T(N)} = \frac{T_s+T_p}{T_s+\frac{T_p}{N}}$$

Esta expresión se denomina **Ley de Amdhal** [59]. La expresión común de esta ley es la desigualdad ya que el *Speed-Up* tal y como aparece en esta expresión es el **mejor** que se puede obtener. En un escenario real siempre será inferior o igual, pero nunca superior.

La ley de Amdhal se usa para evaluar la conveniencia de la paralelización de un código [71]. Si la fracción secuencial del mismo no es mucho más pequeña que la fracción paralelizable, entonces el speed-up no proporcionará la ganancia que compense la paralelización. Aún así la ley de Amdhal es demasiado optimista ya que no considera mucho de los aspectos que degradan el rendimiento en paralelo.

Una definición más profunda del *Speed-Up* aconseja la inclusión de, como mínimo, dos tiempos a tener en cuenta:

- T_s, tiempo original de la versión secuencial en un procesador
- T_{is}, tiempo medio adicional que se invierte en tareas propias de la paralelización como comunicaciones, configuración, etc... Este tiempo puede llegar a depender de N, la pre-

sunción más simple indica que cada procesador debe gastar todo este tiempo, así que podemos considerar que el tiempo adicional secuencial para este ejemplo es $N * T_{is}$

- T_p El tiempo en ejecutar la fracción paralelizable en la versión secuencial.
- T_{ip} El tiempo medio adicional que cada procesador invierte en realizar las labores de configuración y computación en paralelo, este tiempo puede incluir el tiempo en que el procesador está ocioso.

Es importante notar que la contribución más significativa al tiempo T_{is} viene determinado por el tiempo en comunicar sub-tareas paralelas (figura 1.3).Este tiempo siempre se encuentra presente, pues aún en las tareas más simples e independientes es necesario que sobre ellas se ejerza algún tipo de control y este se hace con mensajes a través de la red. Este tiempo puede jugar un papel muy perjudicial en la *escalabilidad* de la solución paralela. Este tiempo T_{is} es el responsable de toda la investigación invertida en la optimización de las comunicaciones y en el hardware.

Por otro lado, suele ser una norma de uso aconsejado el combinar los tiempos T_{ip} y $N * T_{is}$ en una expresión $T_o(N)$ (tiempo de sobrecarga -*overhead*-) que recoge el tiempo empleado por cualquier evento, tarea, en definitiva cualquier aspecto relacionado con la sobrecarga debida a la computación paralela y que complica el escalado hasta el factor de N. Esta descripción, aún, puede considerarse simplificada pero recoge con fidelidad el perfil común de la sobrecarga debida a las tareas de paralelización.

Con todo lo establecido hasta ahora, es posible establecer una definición formal de *Speed-Up* :

$$S(N) = \frac{T_s+T_p}{T_s+N*T_{is}+\frac{T_p}{N}+T_{ip}}$$

Esta expresión es lo bastante descriptiva como para hacerse a la idea de las propiedades de escalado de un aplicación que se pretende paralelizar para ser enviada a un cluster.

Como conclusión de este punto es posible afirmar que la Ley de Amdhal es una interesante herramienta a la hora de determinar el grado de paralelismo viable en una aplicación. La ley de Amdhal es una medida indirecta que recoge las características de la aplicación, su patrón de ejecución, el uso de las comunicaciones (factor que depende de la metodología escogida a la hora de implementar la aplicación y el impacto de estas en la computación. La ley de Amdhal

proporciona una información excepcional, por tanto, para determinar la *eficiencia* ($\frac{S(N)}{N}$, donde S(N) es el speed-up y N el número de procesadores) de una aplicación paralela en un determinado entorno computacional.

Antes de avanzar, es preciso notar que la aparición de las arquitecturas multicore no sólo ha revolucionado los paradigmas de desarrollo y modelado de aplicaciones sino también la forma de medir el rendimiento o beneficios de tal paralelización. Dos autores, Marty y Hill [20], usando la ley de Amdahl tal y como la conocemos actualmente arrojan datos negativos y pesimistas sobre la escalabilidad de las arquitecturas, en esta línea se encuentran también a autores como Samuel K. Moore [72] que acota el efecto de los multicores a las aplicaciones que procesan grandes volúmenes de datos. En [73] (donde se hace una revisión sintetizada y muy interesante a la ley de Amdahl), se presenta una interesante reescritura de la ley de Amdahl para esta arquitectura (y en general, para todas). Los resultados obtenidos en el trabajo de X. Sun e Y. Chen , indican que la escalabilidad en estas plataformas es extensiva. Por otro lado, los postulados de la ley de Gustafson [60] no parecen haber sido perjudicados con la aparición multicore. Gustafson trata de aliviar el peso de la ley de Amdahl (sección secuencial) incluyendo más computación al incluir más nodos. Aspecto que encaja con las nuevas arquitecturas.

1.1.4. Aplicaciones paralelas y científicas

Una aplicación científica se caracteriza por sus exigentes demandas de recursos de la arquitectura en la que se ejecutará. Abordar un problema computacional que requiere un uso intensivo y extenso de los recursos obliga a implementar algoritmos paralelos que puedan repartir el *peso computacional* entre diferentes *entidades de procesamiento*, los trabajos [74], [75] y [76] pueden servir como referencia. La posibilidad de paralelizar algoritmos secuenciales, está restringida a un subconjunto de estos (*Ley de Amdhal*). Es evidente que la paralelización de un algoritmo secuencial puede abordarse desde distintas perspectivas (no siendo todas igualmente eficaces). El éxito de la aplicación paralela depende de cómo se abordó la paralelización y de la idoneidad de la técnica para la arquitectura sobre la que se espera ejecutar la aplicación. La técnica tradicional seguida para la paralelización de aplicaciones científicas ha sido *divide and conquer* que consiste en hacer divisiones del problema original hasta que se llega a obtener subtareas fácilmente resolubles (la división se debe hacer con cierta precaución para evitar dependencias entre subtareas que generen demandas de comunicación ya que empeoraría la *eficiencia*).

Una aplicación paralela por tanto se compone de un conjunto de tareas susceptibles de ser ejecutadas en *entidades de procesamiento disjuntas*. La comunicación necesaria (dadas las lógicas dependencias que entre estas subtareas han de existir) se han implementado, en la mayoría de los casos, usando librerías de paso de mensajes como MPI [77] PVM [78] y similares (en [48] se puede encontrar una revisión bastante útil sobre los modelos de programación cuando se usa MPI o bien PVM, concretamente en el capítulo 3 de la primera parte del volumen). De esta forma la aplicación secuencial científica se modifica en dos aspectos:

- Divisiones de tareas (comúnmente siguiendo un modelo SPMD).
- Inserción de directivas de comunicación para mantener la secuenciación entre tareas.

En tales aplicaciones es una tarea compleja coordinar o sincronizar las diferentes instancias de ejecución. Las aplicaciones de esta clase han adoptado la naturaleza *iterativa* como aproximación para obtener un sincronismo implícito. Sincronismo que facilita el control de la computación en un modelo maestro/esclavo.

Así pues se puede decir que las aplicaciones científicas que se han desarrollado hasta la fecha, en un gran porcentaje tienen características comunes :

- Desarrolladas en lenguajes como C o Fortran.
- Iterativas
- Paso de mensajes para compartir la información.
- Muchas de ellas implementan problemas de naturaleza irregular. Otras no contemplan que al distribuir las tareas entre diferentes procesadores, las iteraciones pueden diferir considerablemente en tiempos de ejecución.

1.1.4.1. Naturaleza irregular de las aplicaciones científicas

Las aplicaciones científicas implementan complejos modelos matemáticos que caracterizan al problema en estudio, sirvan como ejemplo aplicaciones como [79], [80], e incluso se desarrollan artificios para amortiguar el efecto de la irregularidad de las aplicaciones [81]. La realidad es que el modelo matemático constituye la forma de reflejar el comportamiento del problema pero no aporta información sobre las estructuras de datos con las que alimentar a la aplicación. Los lenguajes de programación tradicionales, salvo honrosas excepciones, poseen estructuras de

datos generalistas y que en ocasiones no reflejan ni el dinamismo de los datos que se manejan en el modelo ni la irregularidad de los mismos. En [82] citan esta problemática pero la abordan desde una perspectiva centrada en resolver conflictos entre hebras, mucho más bajo nivel de lo que pretendemos en este trabajo de tesis, pero se apoya en los mismos conceptos. Como contraste a los lenguajes de programación tradicionales se pueden encontrar hoy lenguajes como SBASCO [83] donde se aprovechan componentes software y esqueletos software para desarrollar aplicaciones paralelas.

Una de las características que abanderan un gran porcentaje de aplicaciones en el campo de la ciencia y de la ingeniera es la irregularidad, en muchas ocasiones debida al dinamismo de los datos, en otras a lo indeterminista del algoritmo. Paradójicamente las estructuras de datos que se emplean en estos algoritmos para nutrir de información al modelo suelen ser poco flexibles (hasta el punto que se han creado algoritmos para tratar matrices dispersas, por ejemplo). El modelo de paralelismo impuesto por las aplicaciones científicas y paralelas es tal que la división de tareas propuesta necesita su *porción* de la estructura de datos. Una estructura de datos regular y poco flexible, como una matriz, que se usa para modelar los datos irregulares y no uniformemente localizados de una aplicación científica, invita a que se divida de igual forma y se reparta equitativamente generando desequilibrios en las operaciones de cálculo.

Por otro lado y este punto no atañe a la naturaleza de la aplicación aunque si a la forma de implementarla, la elección de un lenguaje como Fortran o C y C++ hacen bastante difícil reparar las situaciones de computación irregular que pueden afectar a la aplicación por causas externas, como por ejemplo compartir la CPU de uno de los procesadores de cálculo con más subtareas, de la misma o de otra aplicación.

1.2. Arquitecturas paralelas

La computación paralela es un modelo eficiente para reducir los tiempos de cálculo de aplicaciones que tienen una elevada carga computacional. Los algoritmos paralelos ejecutados sobre una arquitectura paralela obtienen un mejor rendimiento cuando:

- Se tiene un conocimiento profundo de la aplicación.
- El programador es capaz de explotar el paralelismo de la aplicación, especialmente el paralelismo de datos [28].

- Se tiene un buen conocimiento de la arquitectura sobre la que se trabaja.

Un computador paralelo se compone de un conjunto de N procesadores que colaboran en la resolución de un problema. El usuario espera que el tiempo de ejecución obtenido en un computador paralelo se ajuste a $1/N$ *veces* el tiempo de ejecución en un procesador. Esta aceleración o ganancia está en función de muchos aspectos, entre los más importantes se cuentan los tres aspectos citados antes, ya que por ejemplo, de una distribución o paralelismo de datos erróneo se derivan comunicaciones que penalizan el escalado de la aplicación. Esta sección repasa brevemente las arquitecturas paralelas existentes y se centra en las arquitecturas MIMD (*Multiple Instruction Multiple Data*).

1.2.1. Jerarquías de memoria

La memoria principal en una arquitectura paralela no sigue el mismo patrón (ni de diseño ni de acceso por cuestiones evidentes de rendimiento) que en las arquitecturas genéricas como las estaciones de trabajo o los PC normales. En estas arquitecturas la memoria es o bien *memoria compartida* (el espacio de direcciones es común a todas las entidades de procesamiento) o *memoria distribuida* (donde cada entidad de procesamiento posee su propio espacio de direcciones local). Una tercera opción recoge la *mezcla* de las dos primeras, *memoria distribuida y compartida* en la que cada entidad de procesamiento posee su espacio de direcciones local y además tiene acceso a un área de memoria que no pertenece a ninguna entidad de procesamiento y que a la vez pertenece a todas, el acceso a la memoria local es más rápido que el acceso a la memoria compartida, en esta alternativa.

Aquellas arquitecturas en las que a toda la memoria principal se accede con la misma latencia y el mismo ancho de banda se las denomina sistemas *UMA (Uniform Memory Access)*, típicamente son los sistemas de memoria compartida. Un sistema que no posee esta propiedad se denomina *NUMA (Non Uniform Memory Access)*, esta propiedad suele caracterizar a las arquitecturas de memoria distribuida.

Caches

Las caches son memorias de pequeña capacidad. Son extraordinariamente rápidas y están situadas estratégicamente cerca del procesador. Su uso se apoya en la explotación de los conceptos de *localidad espacial y localidad temporal*. Las arquitecturas paralelas presentan ciertos

problemas con las caches que deben almacenar en más de un lugar el mismo valor. Estas arquitecturas han de poseer un sistema de coherencia de caché para evitar incoherencias en la ejecución de un programa. La coherencia de caché traza los valores que los distintos procesadores han de tener en sus caches y elimina los valores incorrectos, asegurando que la ejecución del programa en todos los procesadores es correcta y coherente. El método más conocido para mantener a la caché a salvo de incoherencias es el *bus snooping*. Diseñar un sistema de coherencia de caché para una computadora paralela de memoria distribuida es una tarea compleja y es precisamente uno de los motivos que impiden que estas arquitecturas escalen con propiedad. En contraste, las arquitecturas distribuidas no tienen este problema.

Comunicación

La comunicación *procesador-procesador* o la comunicación *procesador-memoria* puede implementarse en hardware desde varias perspectivas, incluyendo la memoria compartida (ya sea multiplexada o multipuerto), conmutador de barras cruzadas (crossbar switch), bus compartido, red de interconexión (existen miles de topologías como la estrella, el anillo, el árbol, el hipercubo o las mallas n-dimensionales). Las computadoras paralelas basadas en redes de interconexión precisan de algún tipo de *tabla de rutas* para permitir el envío de mensajes entre nodos que no estén directamente conectados.

Las computadoras paralelas de memoria distribuida, en este aspecto presentan una desventaja cuando son comparadas a las computadoras paralelas de memoria compartida.

1.2.2. Arquitecturas SIMD, MISD, MIMD

Las arquitecturas paralelas con las que se trabaja, se pueden clasificar en cuatro bloques fundamentales, a saber SIMD, MISD y MIMD.

SIMD

SIMD -Single Instruction Stream, Multiple Data Stream- En una máquina *SIMD*, varios elementos procesados se supervisan por una unidad de control. Todas las unidades de procesamiento reciben la misma instrucción desde la unidad de control pero operan con diferentes conjuntos de datos, los cuales provienen de distintos flujos de datos. Las características principales de este tipo de arquitecturas son

- Distribuyen el procesamiento sobre una larga cantidad de hardware.
- Operan concurrentemente con muchos elementos de datos diferentes.
- Realizan el mismo cálculo en todos los elementos de datos.

Cada unidad de procesamiento ejecuta la misma instrucción al mismo tiempo, y los procesadores operan de manera síncrona. El potencial de *Speed-Up* de las máquinas SIMD es proporcional a la cantidad de hardware disponible. El paralelismo hace que las máquinas SIMD desarrollen altas velocidades.

Algunas características generales de las computadoras SIMD son:

- Menos hardware debido a que usan sólo una unidad global de control.
- Menos memoria ya que sólo se necesita una copia de las instrucciones colocada en la memoria del sistema.
- Flujo de instrucciones simples y sincronización implícita del procesamiento de elementos, lo que hace que una aplicación SIMD sea comprensible, fácil de programar y fácil de trazar.
- Instrucciones de control de flujo y operaciones escalares que son comunes a todos los elementos procesados que pueden ser ejecutados en la unidad de control, mientras los procesadores están ejecutando otras instrucciones.
- Necesidad de los mecanismos de sincronización sobre los procesadores, después de cada ciclo de ejecución de una instrucción.
- Menos coste ya que sólo se precisa un decodificador en la unidad de control.

La computadora ILLIAC IV que se desarrolló en la Universidad de Illinois en el campus de Urbana-Champaign es un ejemplo de este tipo de arquitecturas.

MISD

Multiple Instruction Stream, Single Data Stream, las máquinas paralelas encuadradas en esta clase pueden ejecutar varios programas distintos con el mismo elemento de datos. La arquitectura puede describirse en dos categorías:

- Una clase de máquinas que requieren de distintas unidades de procesamiento que pueden recibir distintas instrucciones para ser ejecutadas con los mismos datos. Sin embargo, este tipo de arquitecturas es más un ejercicio intelectual que una configuración práctica.
- Una clase de máquinas tales que el flujo de datos circula sobre una serie de elementos de procesamiento. Las arquitecturas *pipeline* tales como los arrays sistólicos entran dentro de este grupo de computadoras.

MIMD

Multiple Instruction Stream, Multiple Data Stream. Un sistema MIMD es un sistema multiprocesador o una multicomputadora en donde cada procesador individual tiene su unidad de control y ejecuta su propio programa. Las computadoras MIMD tienen las siguientes características:

- Distribuyen el procesamiento sobre un número independiente de procesadores.
- Comparten fuentes, incluyendo el sistema de memoria principal, sobre los procesadores.
- Cada procesador opera independientemente y concurrentemente.
- Cada procesador ejecuta su propio programa.

Las arquitecturas MIMD pueden ser, a su vez, divididas en dos grandes grupos, las arquitecturas de *memoria compartida* y las arquitecturas de *memoria distribuida*.

En la arquitectura de *memoria compartida*, los procesadores del computador paralelo comparten un único espacio de direcciones de almacenamiento que es accesible por todos los procesadores mediante una red de interconexión. El problema que presentan estas arquitecturas frente a las de memoria distribuida es la escalabilidad. En aras de solucionar este problema se ha trabajado en varias direcciones :

- Incorporar caches en el sistema de memoria, de forma que cada procesador disponga de una caché local.
- Usar redes de interconexión de menor ***latencia*** y mayor ancho de banda que sustituyan al bus.
- Distribuir físicamente los módulos de memoria entre los procesadores.

Estas líneas de trabajo dan como resultado una división en la clasificación dentro de estas arquitecturas:

- Arquitecturas de procesamiento simétrico. La principal característica de estas arquitecturas, también denominadas *SMP* es que el tiempo de acceso a memoria es el mismo para cada procesador, independientemente de la posición física donde se encuentre, por este motivo estas arquitecturas reciben el nombre de arquitecturas *UMA*.

  Las arquitecturas que usan *bus* un como topología de red de interconexión tienen una mayor relación coste-eficiencia para un número bajo de procesadores, por ejemplo, entre 8 y 16 procesadores. Mientras que sistemas que usan redes de interconexión más complejas, pueden escalar hasta un mayor número de procesadores. Algunos ejemplos de estas arquitecturas son los nuevos servidores bi-procesador y cuatri-procesador que nos ofrecen las principales casas de procesadores y también los novedosos multicore de Intel y AMD.

- Arquitecturas de memoria compartida-distribuida, se caracterizan por el hecho de que el tiempo de acceso a la memoria no es el mismo para todos los procesadores sino que depende de la posición física donde se encuentre. Por esta característica, se denominan arquitecturas *NUMA*. La principal ventaja respecto a las arquitecturas *UMA* es que estas son más fácilmente escalables, llegando a varios cientos de procesadores en un único computador (el mejor ejemplo de esta arquitectura es el *Stanford DASH* con 64 procesadores MIPS-R3000/3010 a 33 MHz. Como se advirtió en una sección previa, las arquitecturas *NUMA* deben mantener la caché actualizada en todos los procesadores, para esto habilitan protocolos de coherencia. Las arquitecturas que poseen tales protocolos se denominan *cc-NUMA* (coherent cache NUMA).

En la arquitectura de *memoria distribuida* cada procesador tiene su propio espacio físico de direcciones de memoria local a la que no pueden acceder directamente otros procesadores del computador. Las ventajas de este modelo frente al de memoria compartida son :

- Cada procesador puede usar todo el ancho de banda para el acceso a su memoria local.
- Es más escalable; el tamaño del sistema sólo está limitado por el tipo de red de interconexión usada para conectar procesadores entre sí.
- No hay problemas de coherencia de caché.

La programación de este tipo de sistemas es más costosa, ya que el intercambio de datos entre procesadores se hace mediante el paso de mensajes a través de una red de interconexión. Debido a esta característica, estas arquitecturas también se denominan *de paso de mensajes*. *Mare Nostrum* el supercomputador que opera en el *BSC* (Barcelona Supercomputing Center) es un ejemplo de esta arquitectura.

1.2.3. Evolución e impacto del hardware en la escalabilidad de las aplicaciones

Desde el nacimiento de los procesadores, allá por la década de los 70 los microprocesadores han implementado el modelo de computación convencional diseñado por *Von Neumann*, con escasas modificaciones. Para un usuario el procesador es una entidad que ejecuta instrucciones una tras otra y que está conectado a una memoria monolítica que contiene todos los datos que el programa necesita. Debido a la conveniencia de la compatibilidad hacia atrás cada diseño se ha visto lastrado y obligado a mantener esta abstracción. Desde la perspectiva de la memoria, se ha terminado diseñado caches de más capacidad (para mantener las porciones de memoria accedidas con mayor frecuencia en pequeñas porciones de memoria más cercana al procesador) y ficheros de registros, más grandes, capaces de mantener más datos activos en un área de memoria excepcionalmente pequeña y gestionada por los compiladores. Desde la perspectiva del procesador, esto ha llevado a modificaciones destinadas a conseguir uno de los siguientes objetivos:

- Incrementar el número de instrucciones capaces de ser *emitidas* en cada ciclo.
- Incrementar la frecuencia de reloj por encima de lo que la *Ley de Moore* augura.

Las técnicas de pipelining (*cauces*) ha permitido a los diseñadores de procesadores, incrementar la frecuencia de reloj (cada vez se necesita usar por instrucción menos lógica para cada etapa).

Los procesadores *superescalares* (ver fig. 1.1(b)) se diseñaron para poder ejecutar múltiples instrucciones en cada ciclo (provenientes de un sólo flujo de control), se usa la reordenación de instrucciones para eliminar las dependencias de datos y así poder aprovechar la ejecución en paralelo.

Ambas técnicas han progresado porque permiten que las instrucciones se ejecuten más rápido (a la par que dan la ilusión que todo se está ejecutando en orden y secuencialmente). El rendimiento obtenido con estas técnicas hardware puede ser incrementado si *los compiladores*

ajustan las secuencias de instrucciones y el diseño de los datos con los que trabajan [84] para que se adapten más a la arquitectura (pipelined o paralela) subyacente. Pero lo realmente importante es que nuestro código se podrá ejecutar correctamente en nuevas arquitecturas y con cierta ganancia (aunque no la máxima) sin experimentar ninguna modificación.

Como se ha adelantado, se está convirtiendo en una tarea compleja seguir explotando estas técnicas para incrementar el rendimiento de los procesadores modernos. Los flujos de instrucciones típicos tienen un paralelismo máximo entre instrucciones. Los procesadores superescalares capaces de emitir más de cuatro instrucciones por ciclo consiguen extraer muy poco rendimiento extra de las aplicaciones.

Para complicar más las cosas, construir procesadores superescalares capaces de explotar algunas instrucciones más por ciclo, resulta excesivamente caro ya que la complejidad de la lógica para encontrar instrucciones paralelas, dinámicamente, es proporcional al cuadrado del número de instrucciones que pueden ser emitidas en simultáneamente. De manera análoga, un cauce con más de 10-20 etapas hace que las etapas en las que se ha de dividir una instrucción sean demasiado cortas, obligando a usar por etapa infimamente la lógica del cauce. Además de la complejidad añadida y sobrecarga en el cauce provenientes de añadir registros extras para las etapas y mecanismos de *bypass*.

Los avances tanto en técnicas superescalares como en pipelining están además limitados por el hecho de que necesitan cada vez más transistores integrados en la lógica de cada procesador, haciendo que el coste de control y verificación de tales procesadores no compense la ganancia que de ellos se pueda esperar. A esto hay que añadir que el desarrollo en el progreso de los procesadores convencionales se ha detenido, el causante: *el consumo* que ha ascendido -a la par que las técnicas de pipelining y superescalar- de menos de un vatio a cerca de los cien. Desafortunadamente a esto se le suma que la capacidad de *disipación de calor* no escala exponencialmente con la misma facilidad que los requisitos de consumo (contrastando el incremento de consumo con la capacidad de disipación, puede comprobarse que los procesadores de la década de los 80 apenas poseían disipadores hasta los actuales que poseen grandes sistemas de refrigeración, de seguir evolucionando acorde a la *ley de Moore* se precisarían sistemas de refrigeración por agua, e incluso hidrógeno).

El límite práctico que han encontrado los procesadores convencionales a su evolución en la ganancia de rendimiento se ha visto impuesto por la combinación del limitado paralelismo de

instrucciones hábil para las técnicas superescalares, los límites del cauce (en etapas) y el binomio consumo-disipación.

A pesar de que la caché todavía puede ofrecer algunas ganancias de rendimiento (acelerando el acceso a la memoria en el modelo de procesador convencional) lo cierto es que sin el apoyo de mejoras organizativas y arquitectónicas en los procesadores el rendimiento experimenta un tope infranqueable [41].

El *problema de la escalabilidad* (impuesto por la Ley de Amdhal) ha abierto nuevas líneas de desarrollo de procesadores en los que sin exceder las leyes de la física se puede seguir creciendo en rendimiento. El primer ejemplo lo constituyen los procesadores *HT - Hiper hebrados* [41] tecnología que permite que un procesador pueda aparecer ante el Sistema Operativo como si fueran dos procesadores. Las limitaciones del cauce de un procesador (ver fig. 1.1(b) y fig. 1.2) imponen restricciones al paralelismo de instrucciones máximo que un procesador es capaz de alcanzar. La degradación del cauce suele venir, en la mayoría de las ocasiones, condicionada por un fallo de caché o por una predicción de salto fallida. La tecnología *HT* permitía presentar dos hebras de ejecución para aprovechar mejor los recursos hardware del procesador (cuando el cauce se bloquea por dependencias de datos, las unidades funcionales no usadas pueden ser llevadas a la otra hebra de ejecución, evitando tiempos muertos en el cauce).

El siguiente paso ha sido, dados los avances en integración y consumo, unir dentro de un mismo chip (CMP - Chip Multi Processor) [41] a dos núcleos[85] (conjunto de unidades funcionales típicas de un procesador convencional) haciendo que todos los núcleos dentro de un chip compartan la caché (*de nuevo el rendimiento potencial de este diseño se ve limitado, ahora porque la caché es compartida y se generan cuellos de botella en el acceso a la memoria*). La tendencia en el desarrollo de procesadores avanza por la integración de centenas de núcleos en un procesador y en la posibilidad de conectar dos o más procesadores entre sí a través de potentes redes de interconexión.

La tendencia en computación paralela ha seguido la directriz impuesta por la arquitectura descrita en la fig. 1.4(a). Primero se usaron máquinas paralelas compuestas por entidades de procesamiento individuales (*clusters*) conectadas entre ellos con redes de ala velocidad (y baja latencia). Cada entidad de procesamiento consistía en un procesador completo y aislado. El primer paso hacia los Chip Multi Core (CMP) se dio con núcleos que compartían la misma interfaz con el sistema, como se puede ver en fig. 1.4 (4b). Otros sistemas CMPs comparten la

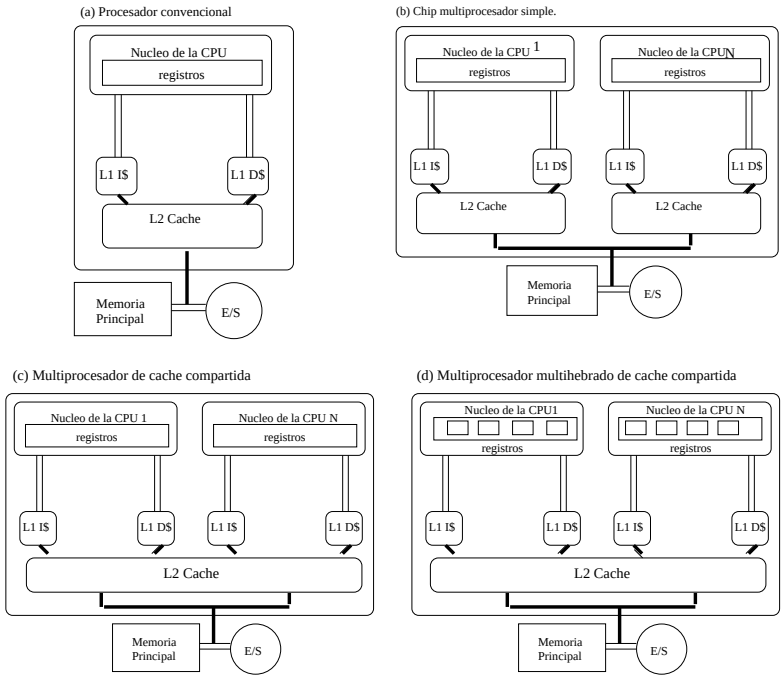


Figura 1.4: Opciones para la implementación de un CMP.

caché (como los representados en fig.1.4 (4c) y (4d) aspecto que permite acelerar las comunicaciones entre dos procesadores sin tener que usar accesos externos al chip. Cada núcleo ha de esperar bastante tiempo a que se satisfagan sus necesidades de acceso a memoria, el siguiente paso lógico en estas arquitecturas ha sido aprovechar la idea que abanderó la tecnología HT, y consecuentemente, se le asignan a cada núcleo varios ficheros de registro para que pueda dar servicio a varias hebras, así mientras un núcleo espera a que la memoria atienda su solicitud, el otro núcleo o incluso hebras del mismo pueden estar atendiendo otras tareas haciendo que el procesador no quede ocioso. Incluso tener un número elevado de hebras en ejecución en el procesador puede incrementar la utilización de los *cauces* que incorporan los sistemas de memoria en los procesadores actuales, ya que el procesador puede solicitar servicios de memoria en paralelo. No obstante, las hebras sufren latencias ligeramente superiores ya que se puede dar el caso de tener a todas las hebras activas y compitiendo por el uso del núcleo en el procesador. La ganancia de poder realizar cómputo en los slots desaprovechados del procesador y la habilidad de solicitar en paralelo numerosas peticiones a memoria compensa esta latencia extra [47].

Las nuevas arquitecturas imperantes, CMP, requieren que las aplicaciones desarrolladas sean o posean naturaleza paralela para poder explotar las arquitecturas multicore. Los modelos de programación tradicionales no se beneficiarán de las nuevas arquitecturas como han venido

haciéndolo hasta ahora [28]. La transición de los sistemas basados en múltiples procesadores a los sistemas multiprocesadores simplifica enormemente los problemas tradicionales de la programación paralela (como minimizar las comunicaciones por la alta latencia de la red). El precio consiste en cambiar la metodología de trabajo [28], [29], en dividir el trabajo en unidades paralelas (sin prestar demasiada atención a la tasa de comunicaciones dada su baja latencia en estas arquitecturas [47]). El reto para explotar estas nuevas arquitecturas está por tanto en los programadores, en los paradigmas de programación y en la paralelización automática. Se hace inevitable pues, adoptar un cambio de paradigma.

1.3. Lenguajes de programación paralela y desarrollo de aplicaciones científicas

Un lenguaje de programación tiene la finalidad de permitir la comunicación con la computadora del mismo modo que el lenguaje natural permite la comunicación entre humanos. La citada analogía es, por naturaleza, limitada ya que con la computadora la comunicación es unidireccional. Se emplea este lenguaje para indicar a la computadora que adopte un determinado comportamiento. Desde esta perspectiva, el lenguaje de programación debería ser entendido como un conjunto de herramientas que se emplean para construir un elemento de ingeniería: un programa. En esta perspectiva, las librerías del lenguaje, los compiladores, y las características especiales del mismo, componen el conjunto de herramientas. De modo que es evidente que la elección del lenguaje de programación afectará al rendimiento global de la aplicación en términos de *eficiencia* y de *calidad*, en el esfuerzo que invertimos en generarlos e incluso en el proceso de desarrollo del software (filosofía). Tal es así que el lenguaje que para una aplicación se considera óptimo, para otra, quizás no lo sea. Por tanto no es de extrañar que encontremos toda una panoplia de lenguajes de programación cuando se plantea el diseño de una aplicación en el campo de la ciencia y la ingeniería.

A la dificultad de la elección del lenguaje y por ende de la metodología, hay que sumar la emergente aparición de arquitecturas paralelas y de arquitecturas multicore-manycore que han dado pie a la aparición de una nueva clase de lenguajes de programación que se añade a todos los que hasta ahora existen [14]. Ejemplos de la importancia en la elección del lenguaje para resolver el problema pueden encontrarse en [86].

1.3.1. Tipos de lenguajes

Los lenguajes más usados para resolver problemas en *ciencia e ingeniería* se pueden dividir en *lenguajes secuenciales* y en *lenguajes paralelos*.

Lenguajes Secuenciales

En las últimas décadas han aparecido multitud de lenguajes (de modelado matemático, de scripting y sobre todo han aparecido muchos lenguajes especializados en áreas especificas).

- **Tradicionales** Muchos de los lenguajes que se usan hoy han sido implementados con C o Fortran. Estos lenguajes tienen asegurada la existencia dada su familiaridad y el excelente soporte que hay desarrollado para sus compiladores.

  - *Fortran* Uno de los lenguajes más usados a la hora de desarrollar aplicaciones científicas y de ingeniería dada su simplicidad y potencia. Fortran es muy rápido, sobre todo por el gran esfuerzo invertido en optimizar sus compiladores. Se inició con la versión 77 y dada su falta de capacidad para *modular* y *reutilizar* código, se creó la versión 90.

  - *C/C++* C apareció para la programación de sistemas y el acceso a características de bajo nivel de manera cómoda (desde alto nivel). Por diversos motivos, entre ellos su versatilidad y velocidad, c fue adoptado por gran parte de la comunidad de desarrolladores, especialmente para aplicaciones no numéricas. C++ conjuga aspectos como la modularidad con la potencia de C. C/C++ ha ocupado, poco a poco, la posición de lenguaje de programación en el área de las aplicaciones científicas. C++ en grandes aplicaciones (y tras todas las optimizaciones invertidas en sus compiladores) resulta ser más eficiente que Fortran. En el trabajo titulado *Modeling biomolecules: Larger scales, Longer durations* [87] se expone un programa de dinámica de moléculas implementado en Fortran, la misma versión del programa en C++ se comporta mejor. Como el rendimiento de C++ en los bucles internos no puede ser mejor que Fortran, se infiere que la supremacía de C++ procede de algoritmos y estructuras de datos más eficientes. La naturaleza de C++ ayuda a los desarrolladores a generar algoritmos más eficientes. De aquí se extrae la conclusión clave de que las abstracciones son de utilidad en la computación paralela y sobre todo ahora con las nuevas arquitecturas.

- *Java* La sintaxis es similar a la de C++. Pero no es un lenguaje ligado a restricciones impuestas por compatibilidad como C++ con C, lo que le infiere más capacidad de abstracción. Es un lenguaje -como C++- orientado a objetos. Java, desafortunadamente, es más lento que C++ (en general) debido a la máquina virtual.

- **Integrados**

  Son lenguajes como Matlab, Maple y Mathematica, entre otros, que han experimentado una gran aceptación en la comunidad científica. Son lenguajes de modelado matemático que se combinan con potentes herramientas para usar gráficos, librerías algebraicas, etc... Su velocidad es menor pues al ser interactivos han de ser interpretados.

- **Específicos** Los lenguajes específicos para un área de aplicaciones científicas, como por ejemplo GPSS que se emplea en simulaciones de eventos discretos, permiten al desarrollador expresarse con mayor facilidad.

Lenguajes Paralelos

La popularidad de las aplicaciones relativas a la ciencia y a la ingeniería se ha visto subrayada por la accesibilidad de las computadoras paralelas. Para conocer el alcance de esta afirmación basta con saber que las aplicaciones científicas computacionalmente intensivas se ejecutaban en máquinas vectoriales como los Cray. Estas computadoras se programaban usando FORTRAN junto con extensiones para operaciones vectoriales.

Cuando las computadoras paralelas dejaron atrás a las computadoras vectoriales, el modelo de desarrollo hubo de ser cambiado. Los desarrolladores tuvieron que descartar o modificar los paradigmas de programación que hasta ahora venían usando.

Los paradigmas que emergieron, en lugar de los viejos, fueron los modelos de memoria compartida, librerías de paso de mensajes, lenguajes donde se explota el paralelismo de datos y todos los paradigmas centrados en datos.

El término *lenguaje de programación* refleja tanto al lenguaje con su sintaxis particular como al compilador que traslada del lenguaje a la codificación binaria. Muchos de los paradigmas de programación actuales no son lenguajes, si se sigue estrictamente la definición. En su lugar estos paradigmas proporcionan librerías con funciones (A.P.I. o Application Programming Interface), sin embargo, usar este tipo de librerías (para escribir programas paralelos) puede llegar a modi-

ficar notablemente el paradigma de programación (el estilo con el que se expresa un *algoritmo paralelo*).En esta sección, por tanto, se considera que las librerías también son lenguajes. Los lenguajes típicamente usados en la programación paralela clásica, pueden catalogarse entre :

- *Espacio de direcciones compartido:* PThreads
- *Paso de mensajes:* MPI, PVM
- *Bucles paralelos(paralelismo de datos):* OpenMP, Compiladores reestructurales,etc...

Los lenguajes que se presentan para los nuevos modelos de programación :

- *Lenguajes orientados a datos*
  - *Hebras:* Multithreading
  - *Objetos:* Charm++, ABC++, CC++, etc...
  - *Manejadores:* Nexus, Mensajes Activos, Converse, etc...
- *Objetos y datos paralelos :* pC++
- *Otros:* Cilk, Linda, multipol, etc...

1. **Programación en memoria compartida (*Shared Memory Programming*)** La programación en memoria compartida es un término amplio para un tipo concreto de modelo de programación destinado a una arquitectura concreta. En este paradigma, el programa del usuario crea procesos (llamados kernel threads o lightweight processes), tantos o más como procesadores disponibles. Todos estos procesos tienen acceso para leer y escribir al mismo espacio de direcciones global, también pueden tener (cada proceso) un área de memoria privada e inaccesible para el resto de procesos. Además de las primitivas de creación de procesos múltiples, en este modelo de programación también se disponen de otras primitivas como las de coordinación, que son *reserva de memoria compartida, bloqueos (locks) y barreras*. Los bloqueos se usan para contener el acceso simultáneo de dos procesos al mismo área de memoria. Una barrera es una primitiva de sincronización. El proceso que invoca a una barrera ha de esperar a que el resto de procesos también la invoquen.

   Las hebras **POSIX** o **PThreads** [88] son una estandarización de este modelo para máquinas Unix/Linux. Con Pthreads los programadores pueden crear más hebras que procesadores

disponibles (donde cada hebra se asocia con un proceso) Para las aplicaciones donde la computación prima, usando este modelo no se deberían exceder el número de procesadores físicos.

Este modelo de programación también puede ser empleado sobre arquitecturas *NUMA*, evidentemente en esta situación la arquitectura de nuevo exige cambios o restricciones al programador, deben evitarse patrones de acceso a memoria remota que sean frecuentes y para leer poco (si es preciso se puede aprovechar cada acceso para traer más información, a la memoria local, que evite lecturas remotas). Para arquitecturas de memoria distribuida se pueden emplear esquemas software de memoria compartida, estos esquemas tienen el inconveniente de tratar de mantener la coherencia de memoria entre los procesadores.

2. **Paso de Mensajes** Se usan en las máquinas de memoria distribuida. MPI se ha convertido en el estándar por excelencia en el paso de mensajes. Contiene más de 100 funciones para usar características de paso de mensajes. Las esenciales permiten realizar *envíos* de datos, *recibir* datos o realizar operaciones de *broadcast (o envío masivo)* de datos entre todos los procesadores. En una arquitectura distribuida, cada procesador ejecuta su propio programa, con su propia memoria. Todos los programas son idénticos (los programas que se ejecutan en todos los procesadores son idénticos) cada programa se comunica intercambiando mensajes (secuencias de bytes etiquetadas).

   El envío de un mensaje consiste en copiar datos del espacio de direcciones del proceso emisor a un búfer. Este búfer -al que se le asocia una *etiqueta*- se envía por la red al procesador destino. Y en el procesador destino se emplea la función *receive* para copiar los datos desde el búfer al espacio de direcciones del proceso adecuado. En esta sencilla aproximación, la llamada a receive queda a la espera de que al procesador destino llegue el búfer. La alternativa asíncrona permite al procesador ocuparse de otras tareas mientras que no llegue el mensaje.

   PVM permite la creación dinámica de nuevos procesos (presumiblemente en nuevas máquinas). Además también permite incluir tipos de datos en mensajes. PVM es especialmente útil cuando el entorno de ejecución es un *c.o.w. -cluster de workstations-* [70].

   El paso de mensajes es un paradigma muy potente y que puede ser considerado de bajo nivel. Los programadores pueden usar el paso de mensajes para construir aplicaciones de

alto rendimiento por que les permite controlar directamente la localidad de los accesos (es decir, pueden acceder exclusivamente a información local mientras que cualquier dato remoto tiene que ser explícitamente referenciado a través de los mensajes). Por otro lado, el paso de mensajes precisa más esfuerzo por parte del programador que los lenguajes de paralelización de datos, ya que el programa debe controlar el tráfico de mensajes y coordinar la acción entre los procesadores.

3. **Paralelismo basado en bucles** La aproximación basada en el paralelismo de datos tiene su origen en una observación: Los cálculos de las aplicaciones de ciencia e ingeniería, a menudo consisten en realizar los mismos cálculos sobre grandes cantidades de datos almacenadas en arrays. Dado que estas operaciones usan bucles para procesar todo el array fueron denominadas metodologías de paralelización de datos o paralelización de bucles.

   En esta aproximación, los usuarios escriben un programa en Fortran, por ejemplo, igual que si lo hicieran en forma secuencial. Para aprovechar el paralelismo basta con incluir directivas del compilador en posiciones estratégicas del programa. Cuando se indica con estas directivas que un bucle es paralelo, el sistema tiene total libertad para ejecutar diferentes iteraciones del bucle en distintos procesadores.

   Ejemplos de este paradigma son OpenMP y HPF.

   OpenMP permite secciones paralelas (donde cada procesador ejecuta la misma sección de código) y paralelismo de tareas. OpenMP permite que todo dato puede accederse desde cualquier procesador (aproximación muy común en máquinas de memoria compartida) OpenMP no suele ser efectivo en máquinas NUMA (suele combinarse con MPI en este escenario).

4. **Nuevas tendencias** Además de los lenguajes descritos, se están comenzando a emplear en el área de la Computación en Ciencia e Ingeniería nuevos lenguajes provenientes de la aplicación de los nuevos modelos y metodologías. Estas nuevas creaciones / lenguajes vienen motivados por varias cosas, entre las que destacan la necesidad de proporcionar soporte para el control irregular y las estructuras de datos irregulares, asegurando la reutilización del software [83] y la necesidad de adaptación a diferentes entornos de ejecución. Durante el último año se están reabriendo muchas lineas de trabajo en lenguajes funcionales [89], [90] por su capacidad para expresar el paralelismo aunque el uso de los mismos es escaso por

la dificultad del modelo de programación.

A menudo las unidades de ejecución en un programa paralelo que se sirve de una arquitectura de memoria distribuida ha de esperar a que se le proporcionen los datos desde procesadores remotos. Ante este escenario, sería deseable que el procesador pudiera realizar otras actividades en lugar de quedar anclado esperando la recepción del dato remoto. Para permitir esto, el proceso ha de tener más de una actividad o subtarea que pueda ejecutar. Si el programador es capaz de expresar computación alternativa, los lenguajes basados en ejecuciones orientadas a los datos (data-driven) pueden mejorar el rendimiento sin sacrificar la modularidad. Las aproximaciones que se citan a continuación activan las entidades definidas por el usuario en función de la disponibilidad de la información / datos (hebras, objetos, manejadores). Son por tanto, aproximaciones basadas en *una ejecución orientada a los datos.*

a) **Multithreading** Esta técnica consiste en la creación de varias hebras de usuario en cada procesador. *Cada hebra ejecuta una función del usuario.* Un inconveniente entre crear una hebra para invocar a una función o invocarla directamente es que la hebra posterga la ejecución de la función. Sin embargo cuando se emplean hebras con paso de mensajes, si una de ellas se dispone a emitir una recepción de mensaje y este aún no ha llegado al procesador, la hebra se detiene y se activa otra hebra que esté disponible para ejecutar. Por otro lado, si la hebra que recibe datos tiene el mensaje listo, entonces la ejecución continúa. Cuando el mensaje llega, el sistema marca a la hebra que lo espera como hebra *activa* para poder planificarla.

El multithreading no es lo mismo que usar hebras de kernel en pequenñas máquinas de memoria compartida. Las hebras usadas en multithreading son hebras de usuario y son ligeras. Suspender una hebra y reanudar otra es una tarea que apenas si consume tiempo. Otra característica es que una hebra de usuario se pausa sólo cuando ella misma lo requiera, en contraste con las hebras de kernel que son suspendidas por el sistema operativo en determinados intervalos.

El propósito del multithreading con hebras de usuario es explotar la concurrencia [91], de modo que de acuerdo con la disponibilidad de los datos (mensajes) se puedan planificar varias tareas en el mismo procesador.

El solape adaptativo entre comunicación y computación que este multithreading nos

ofrece puede ser origen de mejoras sustanciales en el rendimiento. El multithreading, evidentemente tiene limitaciones, entre ellas el uso relativamente elevado de memoria por hebra, cada stack ha de incluirse en el espacio de memoria de cada hebra. El cambio de contexto puede verse afectado por este motivo, llegando a ser más lento que una invocación a una función.

b) **Objetos guiados por datos** Estos lenguajes permiten al programador crear varios objetos por procesador. Un objeto, en este contexto, consiste en un conjunto de funcionalidades (rutinas) que pueden acceder a estos datos. Más de un objeto pueden compartir el mismo conjunto de rutinas, pero cada objeto ha de poseer sus propios datos. Los objetos se discriminan usando un *identificador*. Este *identificador* es útil para determinar el objeto concreto sobre el que se quiere rutinas (métodos).

En un entorno paralelo, se pueden emplear *identificadores* globales. Estos *identificadores* globales infieren independencia del procesador (al objeto). Es posible, por tanto, invocar métodos en objetos remotos gracias al *identificador* global. En cada procesador existe un planificador (scheduler) que se encarga de mantener una cola de *invocaciones (mensajes)* y repetidamente selecciona uno de los mensajes de la cola, identifica al objeto al que se le envía, lo activa y ejecuta el método asociado. El resultado es que se evita que un objeto mantenga bloqueado al procesador en espera de un dato, generando así un **solape adaptativo** entre comunicación y computación.

El primer sistema en usar esta estrategia en máquinas paralelas comerciales fue *Chare Kernel* [92] aunque la idea no parte de los desarrolladores de este sistema, sino que procede de dos proyectos para lenguajes paralelos funcionales (Rediflow y dataflow). Se puede citar a Charm++ [93] como lenguaje que sigue esta filosofía.

La alternativa basada en manejadores (*handlers*), como el caso de los mensajes activos [94] consiste en empaquetar en un mensaje a una función con un parámetro. Este mensaje se envía al procesador remoto. Un planificador se encarga de ejecutar esta función para cada mensaje que llega. Las nuevas implementaciones de esta técnica requieren que cada mensaje contenga un manejador y un puntero de datos global. Este puntero global equivale al *identificador global* de los objetos en la alternativa anterior, por tanto, esta alternativa y la anterior son compatibles.

En esta alternativa, el planificador no precisa crear ninguna hebra para dar servicio

al mensaje, en su lugar es él quien ejecuta la computación asociada a ese mensaje y cuando termina, extrae el siguiente mensaje de la cola. Como resultado, sólo una acción está en curso por procesador y la gestión que antes se precisaba con las hebras, ahora ya no se precisa. Sin embargo, esta facilidad que nos ofrecen los mensajes activos tiene un coste y es la expresividad: Los objetos y los manejadores no pueden detener su ejecución en mitad de la misma por esperar a datos remotos, *a menos que se encuentren dentro de una* ***hebra****.* Es necesario adoptar un estilo de programación por fases donde la petición de datos remotos se encuentre separada del código que se ejecutará cuando el dato llegue.

c) **Lenguajes orientados a objetos. Paralelismo de datos** Estos lenguajes implementan las técnicas de paralelismo de datos en un lenguaje orientado a objetos, como por ejemplo C++. En este caso se aprovecha C++ para crear plantillas de clases que configuran arrays distribuidos capaces de realizar varias operaciones sobre los datos en paralelo. De este tipo de lenguajes podemos encontrar a aquellos cuyo soporte lo ofrece el compilador, por ejemplo es el caso de *pC++* (que se diseñó para realizar en C++, sintácticamente, las mismas operaciones que en *HPF*). *Pooma* [95] y *Overture* son lenguajes que usan mecanismos incluidos en C++ y que aportan excelentes librerías en el campo de las aplicaciones de ciencia e ingeniería.

Cada uno de los lenguajes citados en esta sección (y los que no se han citado, como Erlang o UPC que ultimamente están tomando cierta relevancia) posee cualidades en las que destacan y otras en las que no, redundando en que no existe un lenguaje ideal para cualquier tarea computacional. Esto demuestra la utilidad de sistemas que soporten, en algún modo, la interoperabilidad entre diferentes paradigmas de programación, un ejemplo de estos sistemas es *converse*.

1.4. Computación hebrada y computación orientada a objetos

En [12] se debate un interesante giro que ya no sólo afecta a la forma de estudiar el beneficio de la paralelización sino en al forma de expresarla. Según los autores, la arquitectura multicore está dando pie al uso de diferentes paradigmas de programación y de abstracciones. Tal es así que se revive la discusión sobre el paralelismo implícito (como el explotado por técnicas como el *pipeline parallelism* a través de abstracciones), muy comunes en la década de los 80 y el uso de

los lenguajes declarativos, aunque se subraya que el problema del paralelismo implícito radica en cómo extraerlo eficientemente de los programas, es precisamente en este punto donde autores como Xuli Liu proponen algunas técnicas interesantes basadas en la Orientación a Objetos [96], y en cómo mapear la aplicación en la arquitectura hardware. La otra opción estudiada es el paralelismo explícito donde el programador ha de ser consciente de las características de la arquitectura y de la red de interconexión. Cuando se emplea la programación paralela explícita, se usan modelos poco expresivos como el paso de mensajes (MPI [77], PVM [78]) y se lastra la abstracción perdiendo aspectos tan positivos como la portabilidad [12]. En este trabajo, *Keshav Pingali y David August* discuten la utilidad de las abstracciones frente al uso del paralelismo explícito en el desarrollo de aplicaciones paralelas.

En la sección anterior se expresó el beneficio de usar lenguajes orientados y la conveniencia de que estos se encontraran dento de hebras para poder detener su ejecución a voluntad con el fin de poder obtener un *solape adaptativo* entre computación y comunicación. Esta habilidad tiene un objetivo que es no desperdiciar ciclos de CPU en etapas de comunicaciones. Para explotar este esquema es preciso definir el concepto de hebra, el de objeto y establecer la relación entre ambos.

1.4.1. Características de la programación hebrada. Posix vs. Usuario

Un proceso es la entidad de procesamiento más costosa susceptible de planificación por el núcleo del sistema operativo. Los procesos disponen de recursos que les han sido asignados por el sistema operativo. Estos recursos son memoria, descriptores de archivo, sockets, descriptores de dispositivos, etc... Un proceso no comparte sus recursos con otros procesos, al menos, implícitamente. Los procesos son entidades de ejecución con las que se implementa, a través del sistema operativo, la multitarea de manera apropiativa.

Una hebra o *thread* de ejecución, es la entidad de ejecución menos costosa susceptible de ser planificada por el núcleo del sistema operativo. Es una entidad de ejecución similar al proceso. Ambas representan una secuencia de instrucciones que la CPU debe ejecutar. En general, un *thread* depende de un proceso. Varios *threads* dentro del mismo proceso comparten recursos. Las hebras se reemplazan de manera apropiativa si el sistema de planificación de procesos es apropiativo. Las hebras se caracterizan por no poseer recursos propios, todo lo contrario, los recursos de que disponen son compartidos entre todas las hebras que alberga el proceso. Los

únicos recursos que se pueden pensar propios para una hebra son el *stack*, una copia de los registros y un área para almacenamiento local (*Thread Local Storage -TLS-*) si fuera preciso. En el ámbito de la hebra hay que hacer una distinción entre *hebra de usuario* y *hebra de kernel*. El sistema operativo se encarga de gestionar y planificar a las hebras de kernel. Si las operaciones de gestión y planificación se realizan por librerías ajenas al sistema operativo entonces nos hemos de referir a las hebras como hebras de usuario, comúnmente denominadas *fibras* aunque para el propósito de este trabajo será denominadas *hebras de usuario*. Las hebras de usuario han de pausarse y planificarse para su ejecución de manera explícita.

Concurrencia y estructuras de datos

Las hebras que se encuentran en el mismo proceso, comparten el mismo espacio de direcciones. Esto permite que el código de estas hebras se pueda ejecutar de manera concurrente. La ventaja evidente es que pueden intercambiar datos sin tener que usar los *Inter Process Communication o I.P.C.*. Sin embargo, compartir estructuras de datos alberga un riesgo inherente, y es la posibilidad de que estas puedan ser accedidas por dos o más hebras a la vez para modificar su contenido. Este efecto se denomina *condiciones de carrera*. Se crean APIs que proporcionan primitivas de sincronización como los *mutexes* [97] que establecen bloqueos en las estructuras de datos para hacer seguro el acceso concurrente. En un uniprocesador, una hebra que entre en un *mutex* que se encuentre bloqueado, debe generar el evento de *cambio de contexto* para permitir que otra hebra entre en ejecución mientras que la actual pasa al estado *dormido*. En un multiprocesador la hebra debe consultar el estado del *mutex* hasta que este se libere. En cualquier caso, ambos sistemas minan el rendimiento de la aplicación.

E/S y planificación

Muchas de las librerías que existen para trabajar con el modelo de hebras de usuario se encuentran desarrolladas por completo en el área de usuario del sistema [97]. El beneficio directo de esta aproximación es que el *cambio de contexto* en estas circunstancias es muy rápido pues no tiene relación alguna con el *área de kernel*. El cambio de contexto sólo consistiría en salvar los registros de la CPU que están dando servicio a la hebra en ejecución y cargar el contenido de los registros de otra hebra que deba entrar a ejecutarse. La política de elección de hebra (planificación) al no estar relacionada con el *kernel* puede desarrollarse *a medida* para la aplicación

en cuestión.

En el ámbito de las librerías de hebras de usuario, el principal motivo de pérdida de rendimiento son las llamadas bloqueantes al sistema. Si una hebra de usuario realiza una operación de E/S (implementada con una llamada al sistema bloqueante), la misma quedará a expensas de que termine la operación. El hecho de que la llamada sea bloqueante impide que otras hebras puedan entrar a ejecutarse pues el proceso completo queda bloqueado por el *kernel* dejando sin opción de ejecución al resto de hebras del proceso. La solución a este inconveniente pasa por la creación de llamadas de E/S al sistema que sean no-bloqueantes y que permitan que otras hebras puedan estar activas dentro del proceso mientras una de ellas espera a que esté disponible la información de la operación de E/S.

Algunos ejemplos de librerías de hebras de usuario disponibles en sistemas operativos de renombre, son :

- Win32, proporciona una API para trabajar con su versión particular de hebras de usuario.
- SunOS 4.x implementó en el área de usuario el concepto de LWP (light weight process) denominándolo *green threads* que son el modelo de hebras seguido por Java.
- NetBSD 2.x y DragonFly BSD hicieron lo mismo que SunOs 4.x.
- SunOS 5.2 a SunOS 5.8 implementaron un modelo de dos niveles, multiplexando una o más hebras de usuario sobre una hebra de kernel.

El uso de hebras de kernel, simplifica el código del usuario ya que es el núcleo del sistema operativo el que se encarga de los detalles más complicados de las tareas de multithreading. Sin embargo, en los uniprocesadores, el uso de hebras de kernel puede ocasionar un cambio de contexto sin que se haya previsto dando lugar a condiciones de carrera y a complicados errores inherentes a la concurrencia. Situación que se complica si el procesador es SMP.

Modelos de implementación

En la figura 1.5 pueden observarse varios escenarios de operación con hebras. El primer escenario lo representa el *proceso 1*, modelo de proceso tradicional, donde existe sólo una hebra de control y esta se hace corresponder con una entidad de ejecución planificable por el kernel.

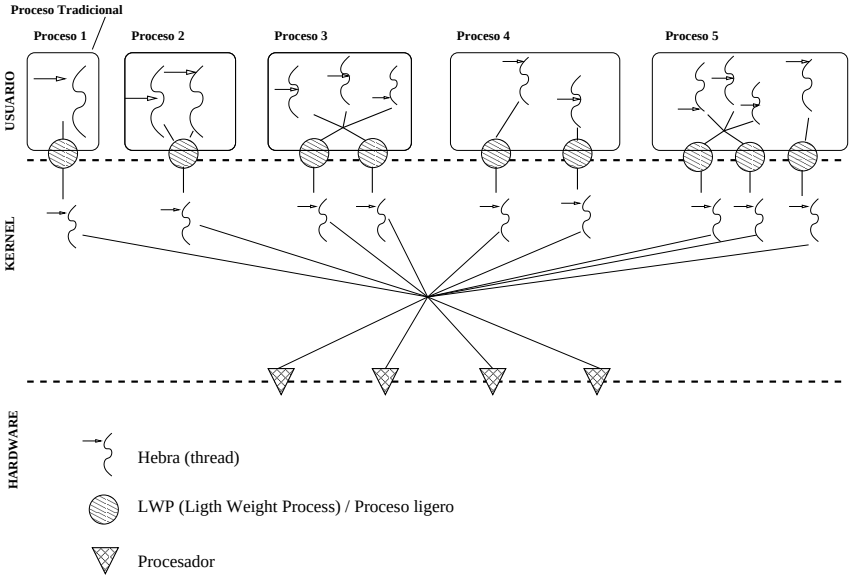


Figura 1.5: Hebras hardware, proceso ligero y hebras de usuario.

El *proceso 2* muestra el ejemplo claro del uso de hebras de usuario, donde las hebras de la aplicación han de ser planificadas enteramente desde el espacio del usuario. El kernel desconoce por completo la existencia de tales hebras.

Los *procesos 3, 4 y 5* son ejemplos de un sistema multihebrado, donde la librería de usuario es responsable de elegir una hebra de usuario para que pueda ser asociada a una de las unidades planificables por el kernel. Este sistema disfruta de un cambio de contexto muy rápido pues no se precisan llamadas al sistema. Pero su implementación es extremadamente compleja pues debe coordinar el área de usuario con el área de kernel.

1.4.2. Relación entre la programación hebrada y la Orientación a Objetos

La programación orientada a objetos parte de la premisa de poder tener en el seno de una aplicación, varios objetos activos. La naturaleza de los mismos y su patrón de comportamiento pertenecen a la semántica de la aplicación. Lo cierto es que entre el concepto de objeto y el concepto de hebra existe un cierto paralelismo que se puede aprovechar.

Como se observa en la figura 1.6, la programación hebrada permite sacar partido de la naturaleza concurrente de un código. Ya se ha discutido previamente la ventaja de usar hebras frente a simplemente invocar a la función una vez tras otra. El problema que presenta este modelo es que es poco expresivo. Abordar complejos programas paralelos es difícil si se sigue usando

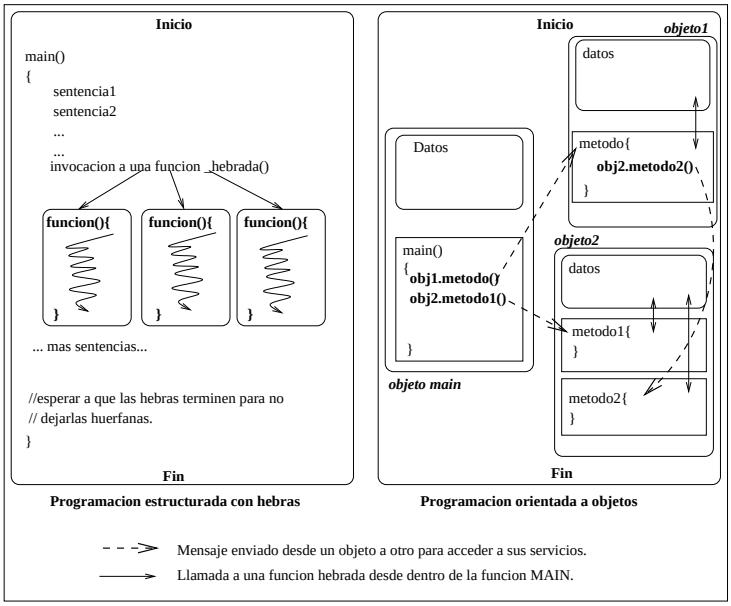


Figura 1.6: Paralelismo en el uso de hebras y objetos.

la programación estructurada. La programación orientada a objetos permite *aislar* en unidades independientes a la funcionalidad y a los datos que esta precisa en entidades de abstracción superior denominadas *objetos*. La computación con los objetos (como se puede observar) depende del flujo de mensajes que entre ellos se envían. El modelo de programación es mucho más útil y expresivo que el anterior. La utilidad de la abstracción redunda en una mejor codificación del problema a su vez más fácil de escalar y mantener. Aparte de los detalles propios de la orientación a objetos, se puede observar que con la orientación a objetos se consigue lo mismo que con la programación estructurada y las hebras, iniciar varias hebras de control de manera concurrente (en orientación a objetos existen métodos atómicos cuya naturaleza impide corrupciones propias de las condiciones carrera y otros efectos adversos de la concurrencia, en caso de estos métodos la concurrencia es de grano más grueso). La necesidad de relacionar ambos paradigmas es dotar a los objetos de la facilidad de hebrar, además, sus métodos, de poder embutir objetos en hebras y controlar su ejecución, etc...

1.5. Adaptabilidad en aplicaciones científicas

Las aplicaciones de ciencia en ingeniería tienen en común que pretenden emular complejos fenómenos para su estudio. La emulación del fenómeno a estudio suele apoyarse en complejos

modelos matemáticos que llevan asociado restricciones de cálculo importantes. En un gran porcentaje de los casos estos modelos matemáticos hacen uso de gran cantidad de información, que difícilmente puede ser procesada en computadoras convencionales. Las exigencias de cómputo y el volumen de datos imponen un sesgo en las computadoras que se pueden emplear. El espectro computacional queda, evidentemente, restringido a las supercomputadoras.

Las supercomputadoras actuales, generalmente, poseen jerarquías de memoria de hasta cinco niveles (cada procesador trabaja con sus registros, caché L1, caché L2, memoria RAM, y memoria RAM externa). Las nuevas arquitecturas se diseñan para que el uso de la caché sea lo más aprovechado posible. Un fallo de caché hace que la aplicación en curso pueda perder bastantes ciclos de reloj en la búsqueda del nuevo dato en RAM. Un buen uso de caché y la capacidad de *ocultar* los accesos a la memoria principal son dos características que pueden hacer perder mucho tiempo a nuestra aplicación.

Generar un marco en que la computación asociada a las aplicaciones científicas pueda llevarse a cabo bajo condiciones aceptables no es una tarea sencilla. Entre otros motivos porque las citadas aplicaciones usan patrones de ejecución difícilmente previsibles. Estos patrones de ejecución no suelen aprovechar las bondades de la memoria caché.

Un amplio porcentaje de aplicaciones típicas en ciencia y en ingeniería se han desarrollado con lenguajes de programación, como C y Fortran. Estos lenguajes son muy estrictos con las estructuras de datos, rigidez que influye mucho sobre el uso de la caché (la *encapsulación* puede mejorar la localidad en las referencias [98]).

1.5.1. La orientación a objetos y las hebras de usuario como medio de adaptación y adaptabilidad.

El diseño de software orientado a objetos ha significado un profundo impacto en la industria de la computación. Conceptos tan elementales como la encapsulación, la abstracción de tipos, el polimorfismo o la herencia se han convertido en la base de la revolución del software convencional y por supuesto forman el núcleo de todas las tecnologías web en Internet, venideras. Las razones para entender esto son simples. El software actual debe interoperar de manera transparente (componentes,...) y debe ser un software totalmente abstraído de la red. Las aplicaciones que se desarrollen con este software han de proporcionar una flexibilidad mayor -en términos de programabilidad- para el usuario final y permitir una especialización extensiva y específica para

cada disciplina.

La necesidad de construir tales productos software con una complejidad cada vez mayor en periodos de desarrollo cada vez más cortos, hizo que los desarrolladores abandonaran su modelo de programación tradicional. De este modo los desarrolladores hubieron de desterrar a los diseños monolíticos y comenzar a crear nuevos sistemas sin más que componer y a extender sistemas de componentes estándar.

Como consecuencia de este proceso surgió el concepto de "middleware". La tecnología de objetos como ActiveX, CORBA, C++ y Java son parte muy importante en esta nueva infraestructura de programación.

Los fabricantes de supercomputadoras han evolucionado de los computadores vectoriales a las máquinas masivamente paralelas, a los sistemas de memoria distribuida, y ahora a los sistemas, a gran escala, de multiprocesadores de memoria compartida y últimamente a los procesadores multicore y manycore. No obstante la tecnología software sigue anclada en los programas escritos en Fortran, tecnología que corría hace veinte años en los Cray-1. Sorprendentemente se ha invertido muchísimo esfuerzo en mejorar los algoritmos sin reparar en el lenguaje subyacente (figura 1.7).

Sin embargo, los responsables del desarrollo de aplicaciones de ciencia e ingeniería también están sometidos a plazos de desarrollo muy exigentes para diseñar programas tan exigentes como los que se solicitan en la industria convencional. Además, la vida de gran cantidad de supercomputadoras es infinitamente más corta que la vida de la mayoría de los aplicativos científicos de gran escala. Por tanto, los responsables del desarrollo de software hubieron de encontrar formas para lograr que el nivel de abstracción y la expresividad de las aplicaciones científicas no se vieran relacionadas con los detalles arquitectónicos de la plataforma. Aunque existen varias formas de conseguir este objetivo, quizás el más natural era el modelo proporcionado por la tecnología de la orientación a objetos.

En 1980 se comenzó con los primeros trabajos sobre este nuevo paradigma. El concepto de abstracción siempre se ha encontrado en dura oposición con el de rendimiento, pero el avance en esta técnica ha reducido la distancia. Esta reducción entre abstracción y rendimiento es precisamente la que está poniendo en evidencia los sistemas de desarrollo monolíticos tradicionales.

Durante toda la evolución han aparecido (y desaparecido) lenguajes O-O generales como Eiffel, Smalltalk, etc y otros de propósito específico como Sather[99] CxC, ... No obstante C++

y Java son dos lenguajes que se han mantenido -hasta la fecha- imbatibles tanto en el desarrollo de software de carácter general como software científico.

El paradigma de la O-O ha sido siempre sugerido como un sistema eficaz para gestionar la complejidad, mejorar la reutilización, y por supuesto como consecuencia directa, mejorar la comprensibilidad de las aplicaciones paralelas. Hay numerosas variantes del paradigma O-O que han sido exploradas en aras de la aplicabilidad a los entornos paralelos [100]. Lo interesante de estas técnicas es que son especialmente atractivas para las aplicaciones irregulares en las que la modularidad y la encapsulación son de utilidad a la hora de tratar con estructuras de datos complejas y con construcciones paralelas. En esta línea se trabaja para la creación de desarrollos de alto nivel basados en sistemas de objetos concurrentes -active object- para que las aplicaciones irregulares adopten los primeros puestos en el ranking de las aplicaciones de alto rendimiento.

La programación orientada a objetos (que usa como uno de sus conceptos clave a la *encapsulación*) ayuda a mitigar la falta de rendimiento en aplicaciones científicas, proporcionada por lenguajes de programación carentes de abstracción [101].

No parece muy productivo reescribir todas las aplicaciones existentes a este nuevo modelo, entre otras cosas porque cuando una aplicación realmente llega a ser eficiente es al cabo de varios años. Si a esto le añadimos el inconveniente de que *el hardware establece las directrices en el software*, encontramos que tenemos aplicaciones científicas desarrolladas en C o Fortran, que quizás no estén haciendo un uso adecuado de las caches o que no están solapando eficientemente las comunicaciones con las computaciones o que de hacerlo, se ha tenido que implementar e medio para llevarlo a cabo. Por otra parte, con las arquitecturas multicore, el cambio ha sido efectivo y se precisa adoptar un nuevo paradigma de programación para explotar a las arquitecturas modernas. Llegados a este punto se plantea la cuestión de qué hacer con las aplicaciones científicas ya existentes.

Según la figura 1.7 podemos comprobar que las aplicaciones de ciencia e ingeniería precisan de arquitecturas adecuadas que reduzcan las exigencias naturales impuestas por los modelos matemático-físicos que implementan. Se ha notado en este capítulo que las arquitecturas han adoptado un giro optando por la integración de varios núcleos en un microprocesador. Por tanto, la relación aplicación - arquitectura se ve alterada. Esta relación aplicación - arquitectura (como se ve en la figura 1.7) hace uso del concepto de paralelismo para poder reducir a tiempos aceptables las ejecuciones asociadas a las citadas aplicaciones.

El paralelismo a implementar y que ofrece rendimiento a estas aplicaciones (y a su relación con las arquitecturas) varía si varía cualquiera de los dos parámetros (aplicación o arquitectura). Si se desea que este paralelismo dote a las aplicaciones de una cierta estabilidad (en términos de reducir la irregularidad intrínseca o extrínseca) es preciso dotar de mecanismos para solapar la comunicación con la computación y dado el caso, de mecanismos para transportar unidades de computación de un procesador a otro. Según lo establecido en este capítulo, estas medidas se pueden alcanzar:

- Solape computación - comunicación: La concurrencia, en este caso es una herramienta efectiva. Las hebras de usuario proporcionan una eficaz solución dada la flexibilidad de poder crear librerías que las gestionen y que actúen sobre su planificación.

- Solución a la irregularidad extrínseca: La migración de unidades de trabajo de un procesador a otro, en este caso, será la solución acertada. La forma de acometer esta tarea es disponiendo de una forma eficaz de agrupar el trabajo en *unidades computacionales*. Esta abstracción la otorga con mucha eficacia el paradigma de orientación a objetos. La capacidad de ocultar el traslado de un procesador a otro se consigue activando en el procesador de origen la ejecución de tareas pendientes. Asociar objetos y hebras, tal y como se observa en la figura 1.7 es un medio eficaz para dotar de esta capacidad al paralelismo que una aplicación de ciencia e ingeniería, precisa.

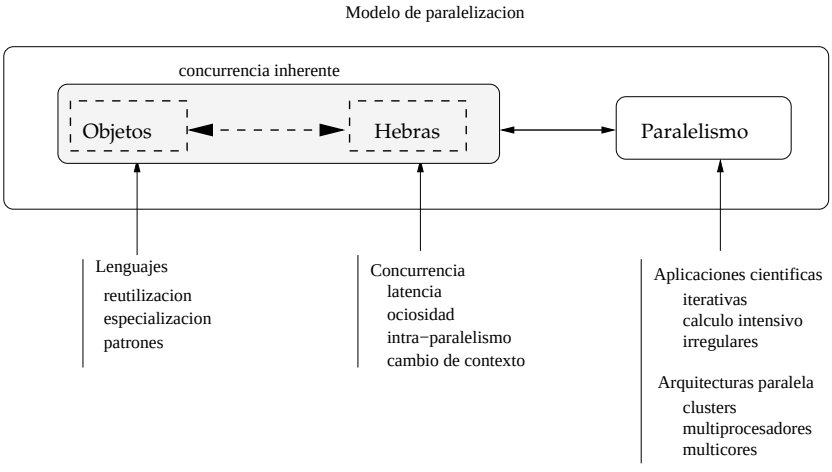


Figura 1.7: Modelo de paralelización.

Esta línea de desarrollo de software es una línea escasamente explotada, existe mucho trabajo de investigación en desarrollar aplicaciones paralelas irregulares que sean eficientes y también

existe mucho esfuerzo investigador empleado en los sistemas de programación concurrente y orientada a objetos.

1.6. Objetivos de trabajo

En este capítulo se ha realizado un estudio del estado del arte y se ha dibujado la situación de desde la que se parte. En este punto quedan establecidos los parámetros de trabajo en los que se centrarán los siguientes capítulos. Por un lado, vamos a trabajar con aplicaciones científicas concretamente con algoritmos de reconstrucción de imágenes tomográficas. Por otro lado, vamos a apoyarnos en plataformas que facilitan el procesamiento paralelo (clusters y multicore). La mayor parte de estas aplicaciones se desarrolla en C junto con librerías de paso de mensaje (como MPI). Si se analiza la ley de Amdahl, estas aplicaciones pueden tener dificultades en cuanto a su escalabilidad y rendimiento:

- El cuello de botella que suponen las secciones secuenciales.
- Las situaciones de desequilibrio de la carga.
- Las latencias, ocultadas manualmente.
- El tiempo de computación efectivo.

Una aplicación desarrollada en C, que se apoye en el modelo de procesos y en el acceso a los servicios del sistema operativo puede experimentar algunas limitaciones:

- La concurrencia de varios procesos en un procesador físico es costosa.
- Las situaciones de desequilibrio de la cargan han de estar muy bien gestionadas por el programador.
- La gestión de las latencias han de ser también muy cuidadas, el límite de la ocultación de las latencias suele venir condicionado por el primer punto.
- Si en este modelo de programación se decide usar hebras, se deberían emplear las del sistema operativo.
- La sección secuencia se convierte en un cuello de botella inevitable.

En esta tesis se pretende favorecer el modelo computacional donde sean aplicables las hipótesis de Gustafson [60], es decir, de ser capaces de hacer escalar el problema a la par que escalamos el número de procesadores, sólo así podremos reducir el efecto de las secciones secuenciales del código. El objetivo consiste en portar aplicaciones de modo que podamos tratar de mejorar de manera directa algunos de los parámetros que limitan la escalabilidad en el modelo de Amdahl. La primera premisa es ser capaz de incrementar la carga para reducir el efecto de la parte secuencial, evidentemente la única forma de conseguir esto es incrementando el número de *programas en ejecución* en cada procesador, en este sentido existen varias alternativas (basadas en abstracciones) que nos resultarán útiles (capítulo 3). La consecuencia directa de esta primera premisa es el tener que gestionar eficientemente la ejecución concurrente de todos estos *programas en ejecución*, de modo que el uso de un *runtime* que ofreza buen soporte de concurrencia se hace más que recomendable. El incrementar la carga de trabajo y gestionar eficientemente la concurrencia hace que el efecto de las secciones secuenciales se va minorado (aunque aparecen otras serializaciones como los accesos a *zonas comunes de memoria* que generen conflictos, de modo que el primer punto sobre el que trabajar será la eliminación de cualquier variable global; para esto el modelo de programación orientada a objetos es excelente. De esta manera podriamos llegar a reducir el *tiempo de computación efectiva*.

La segunda premisa es la de reducir el *tiempo de balanceo*. Lo deseable es que en aplicaciones científicas donde el problema puede generar cargas de trabajo irregulares o bien el propio uso de la arquitectura puede generar situaciones de desequilibrio computacional, podamos corregir el mapeo entre computación y arquitectura de manera adaptativa, esto es, sin tener que tomar decisiones en tiempo de compilación (capítulo 4), para esto es necesario trabajar los siguientes aspectos:

- Estrategias de balanceo de carga diseñadas a medida para nuestra aplicación y su modelo de trabajo, de modo que la carga esté siempre equilibrada sin perjudicar a las latencias (respetando la localidad de los datos).

- Es necesario contar con mecanismos de migración de carga y de computación, eficientes.

- Es necesario hacer que la política de balanceo no intervenga en periodos preestablecidos, sino que actúe cuando la situación lo precise.

La necesidad de revisar el progreso de la computación está justificada cuando la aplicación se ejecuta durante largos periodos de tiempo, y observar el estado de la computación, durante su ejecución sin esperar a que ésta termine para poder analizar las trazas es de interés cuando se está experimentando con una estrategia de balanceo nueva y se desea conocer el efecto directo. En este sentido, el objetivo del capítulo 5 consiste en proporcionar un medio para conectarser visualmente a la aplicación en ejecución.

Capítulo 2

Reconstrucción Tridimensional de Imágenes Tomográficas

2.1. Introduccion

La microscopía electrónica posee un gran potencial como herramienta en el estudio de múltiples problemas estructurales relevantes para la Biología moderna, Biotecnología, Biomedicina y otros campos afines. En concreto, la microscopía electrónica, en conjunción con sofisticadas técnicas de procesamiento de imagen y de reconstrucción tridimensional (3D) [102], permite obtener información estructural cuantitativa sobre la estructura 3D de especímenes biológicos con el fin último de interpretar su función biológica.

En general, estos métodos de reconstrucción 3D se pueden aplicar a un amplio conjunto de especímenes biológicos siguiendo diversas geometrías de adquisición de datos, tales como las llamadas “de único eje de giro”, “de doble eje”, “geometría quasi-cónica”. Las imágenes de proyección obtenidas con el microscopio son combinadas entonces por las técnicas de reconstrucción para obtener una representación 3D de la estructura de los especímenes. Un análisis estructural riguroso requiere que los procesos de adquisición de las imágenes y de reconstrucción 3D sean robustos en el tratamiento del ruido y que proporcionen un nivel de resolución adecuado para permitir una correcta medición e interpretación de las características estructurales obtenidas. Estos métodos, por lo general, son computacionalmente intensivos y requieren tiempos de procesamiento considerables en computadores convencionales.

En microscopía electrónica existen distintas formas de abordar la determinación de la estruc-

tura 3D de los especímenes biológicos. La adopción de una forma u otra depende de la naturaleza del espécimen (su tamaño, su organización en forma de agregados o no, etc). Para especímenes grandes y/o complejos se suele recurrir a la llamada ”Tomografía Electrónica-[103], [104]. Sin embargo, cuando el espécimen es relativamente pequeño, la metodología de reconstrucción se suele conocer como “Microscopía Electrónica Tridimensional”(3DEM) [102].

En tomografía electrónica, una sola muestra de un espécimen es expuesto al microscopio electrónico, obteniendo vistas a diversos ángulos de inclinación siguiendo geometrías de “de eje único de giro”(ver figura Fig. 2.1(izquierda)) o “de doble eje”. El resultado de la adquisición de datos es un conjunto de imágenes (entre 70 y 140) de un tamaño en el rango de 1024x1024 a 4096x4096 pixels. La determinación de la estructura 3D implica combinar ese conjunto de imágenes mediante algoritmos de reconstrucción para generar volmenes del orden de varios Giga Bytes de memoria. Los especímenes susceptibles de estas técnicas se sitúan en el rango subcelular (orden de micras), tales como mitocondrias, dendritas, retículo endoplasmático, virus de gran tamaño, etc. Este tipo de técnicas son directamente comparables a la tomografía en Medicina.

En microscopía electrónica tridimensional (3DEM), las micrografías obtenidas con el microscopio contienen muchas ocurrencias de un mismo espécimen a distintas vistas. De las micrografías se extraen todas esas ocurrencias, obteniéndose decenas de miles de imágenes del espécimen en estudio a distintas vistas. Estas imágenes tienen un tamaño en torno a 100x100 pixels y, en general, verifican la geometría de adquisición “quasi-cónica”. La aplicación de algoritmos de reconstrucción sobre esas imágenes para la determinación de la estructura 3D genera volúmenes del orden del Mbyte de memoria. Los especímenes susceptibles de estas técnicas se sitúan en el rango macromolecular (por debajo de 100 nm).

La determinación de la estructura 3D de especímenes biológicos implica, por tanto, la adopción de distintos enfoques dependiendo de la naturaleza del problema en consideración. No obstante, los algoritmos de reconstrucción son esencialmente los mismos.

Las tendencias actuales en microscopía electrónica hacen uso de técnicas de criomicroscopía con objeto de preservar los detalles estructurales de alta resolución. En criomicroscopía las muestras biológicas se congelan y se exponen al microscopio empleando muy baja dosis electrónica. Como resultado, las imágenes obtenidas del microscopio presentan muy bajo contraste y son extremadamente ruidosas (SNR < 1). Ante estas condiciones, es de vital importancia el desarrollo de algoritmos de reconstrucción que presenten un comportamiento robusto frente al ruido.

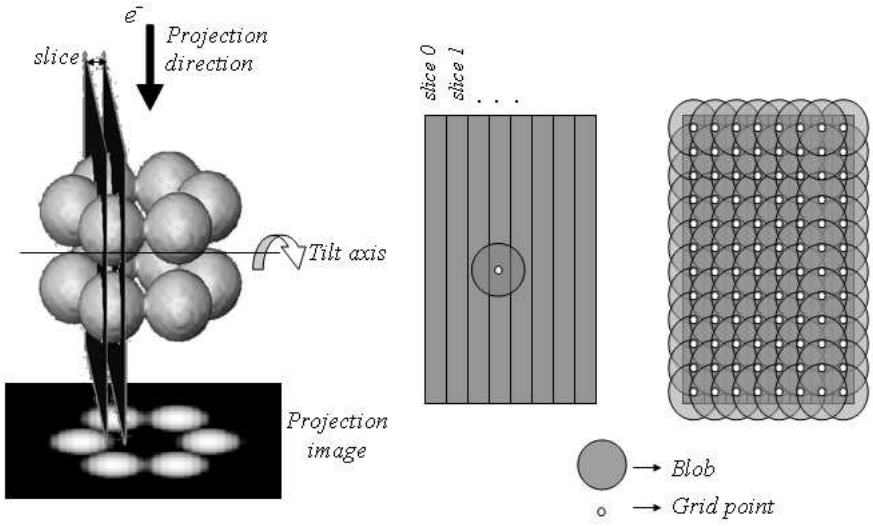


Figura 2.1: Esquema de adquisición de imágenes mediante eje único de giro (izquierda). Esquema de representación de imagen mediante blobs (centro y derecha). Cada columna representa un slice.

2.2. Tomografía Electrónica

La técnicas tomografícas de microscopía electrónica permiten el estudio tridimensional de estructuras biológicas dentro de un amplio margen de tamaños, desde estructuras celulares hasta macromoléculas autónomas. Dependiendo de la naturaleza del espécimen y de la información estructural en estudio se han de emplear técnicas diferentes tanto en la recolección de datos como en la reconstrucción 3D (consúltese [103, 105, 106] para adquirir una perspectiva más documentada).

Este capítulo se centra en la tomografía electrónica de estructuras biológicas complejas (como orgánulos celulares, estructuras subcelulares , o incluso células completas). En estos casos y debido al grosor de los especímenes (50–150 nm) es preciso emplear microscopios electrónicos de alto voltaje (HVEM, 300–500 kVolts) dado el poder de penetración del haz de electrones.

La tomografía electrónica de estructuras celulares posee un gran potencial para eliminar las barreras existentes entre los métodos que proporcionan una resolución cercana a la atómica y las técnicas de microscopía que permiten examinar organismos celulares vivos, tales como la microscopía confocal. La resolución tridimensional que se puede obtener, en la tomografía electrónica de estructuras biológicas complejas, alcanza el rango de los 4-10 nm. Una resolucion que ronde los 2-4 nm permitiría estudiar la organización 3D de componentes estructurales a un

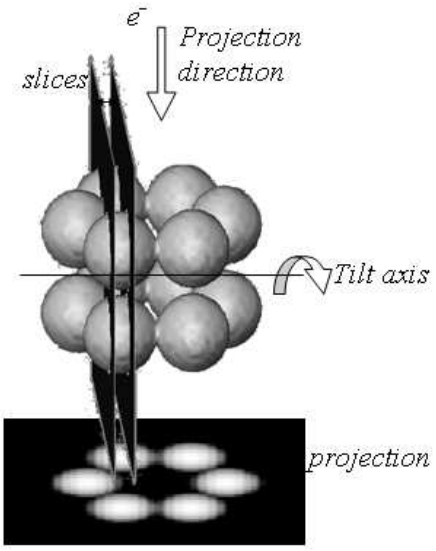


Figura 2.2: Geometría de adquisición de datos en *único eje de giro*. El espécimen se expone al microscopio haciendolo girar en un árco de giro de aproximadamente +/- 60 o 70 grados en pequeños incrementos. Como resultado se obtiene un conjunto de imágenes de proyección, útiles para la determinación de la estructura biológica.

nivel suficiente para poder identificar macromoléculas y sus interacciones en su entorno celular nativo [107].

Tal información relativa a la organización supramolecular y sus interacciones es crucial para comprender la función celular [108]. Se puede, por tanto, encontrar trabajos que avanzan en este sentido [109]. Algunos ejemplos de aplicaciones en el campo de la tomografía electrónica encaminadas a determinar la estructura de componentes celulares son : mitcondria [110], el aparato de Golgi [111], y células completas [112, 113], entre otros.

La geometría de recolección de datos denominada de "único eje de giro.[es] la empleada para recolectar las imágenes (proyecciones) que permitirán calcular la reconstrucción 3D. Para obtener el conjunto de proyecciones, se somete al especimen a la radiación de electrones, haciendolo girar en un rango de +/- 60 o 70 grados en pequeños incrementos de 1-2 grados, la proyección se almacena tras ser capturada por, normalmente, cámaras CCD (Fig. 2.2).

En ocasiones y para cubrir un ángulo mayor se suele repetir la serie de adquisición pero girando al espécimen 90 grados (técnica denominada geometría de doble ángulo de giro)[114, 115]. Los conjuntos de datos típicos en la tomografía electrónica varían de 60 a 280 imágenes. El tamaño de cada imagen suele variar de 1024 x 1024 a 4096 x 4096 pixeles.

La microscopía electrónica impone restricciones, debido a limitaciones físicas, en el ángulo de exposición del espécimen que afectan a la resolución de la reconstrucción. Esta falta de resolución resultante podría solucionarse de disponer de todos los ángulos de exposición en un arco de +/- 90 grados. Esta limitación no está presente en la tomografía médica. Los efectos observables derivados de las limitaciones físicas del aparato de medida pueden observarse en la dirección del eje Z, los detalles estructurales se elongan en este eje. La imposibilidad de "barrer"todo el espectro angular entre 90 grados y -90 grados genera una resolución dependiente de la dirección (anisotrópica), haciendo que los detalles estructurales perpendiculares al eje de giro tiendan a desaparecer.

El ratio SNR en las imágenes de proyección tomadas por el microscopio ha de ser extremadamente bajo, inferior al 0.1. Esto se debe a que la dosis de electrones que ha de atravesar el espécimen ha de ser tolerable por este para así evitar la destrucción de detalles estructurales sensibles.

Como consecuencia de todo esto, la tomografía electrónica de alta resolución aplicada a estructuras subcelulares necesita un método de "Reconstrucción 3D a partir de proyeccionesçapaz de tratar con las condiciones de resolución y distorsión citadas así como con un ratio SNR extremadamente bajo. El método estandard en este área es el método Weighted Backprojection (WBP).

2.3. Fundamentos

La microscopía electrónica de transmisión sólo es capaz de mostrar imágenes bidimensionales (imágenes de proyección) de un espécimen tridimensional. Sin embargo, la combinación de múltiples imágenes bajo ciertas condiciones espaciales y la aplicación de determinados algoritmos computacionales de reconstrucción tridimensional [116, 117], hacen posible la visualización de la estructura tridimensional del espécimen. En este capítulo comentaremos los fundamentos matemáticos y conceptuales que permiten obtener una reconstrucción tridimensional así como los distintos algoritmos existentes.

2.3.1. Imágenes de Proyección

A la hora de hablar de imágenes de proyección hay que diferenciar, para su posterior definición, entre imágenes de proyección de rayos paralelos y no paralelos (cónicos). La geometría

de proyección por rayos paralelos supone que el origen de la proyección se sitúa en un punto impropio del espacio (en el infinito), por lo que los rayos proyectantes se consideran paralelos entre sí. En el caso de la geometría cónica, el origen de proyección es un punto del que parten todos los rayos proyectantes formando una estructura cónica entre sí. En adelante supondremos que la geometría utilizada es de rayos paralelos.

Sea f(x,y) (Por simplicidad elegiremos especímenes bidimensionales), la función que define nuestro espécimen y el espacio que le rodea, definimos la integral de línea:

$$g(\phi, r) = \int_l f(x, y)\, dl \tag{2.1}$$

Tal que ϕ representa el ángulo respecto al sistema de coordenadas utilizado (ver fig.2.3) de un rayo de proyección, r representa la distancia al centro del sistema. Por lo tanto, la función g representa la densidad del objeto acumulada por un rayo al atravesar el espécimen en estudio siguiendo la trayectoria definida por ϕ y r.

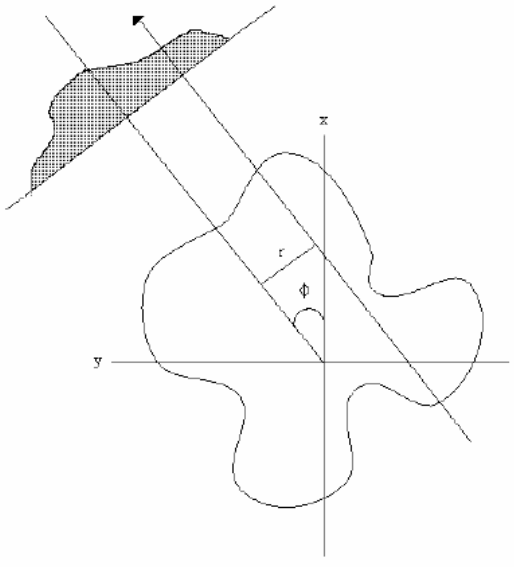


Figura 2.3: Obtención de una imagen de proyección.

Una imagen de proyección es el conjunto de todos los valores de la función g (ecuación 2.1) dado un ángulo ϕ. El conjunto de todas las proyecciones unidimensionales recibe el nombre de *transformada de Radón* del objeto. En el caso que nos ocupa, la reconstrucción 3D, el conjunto de proyecciones es 2D y se denomina *transformada de rayos X* (Natterer 1996; Natterer 1999).

Radon, 1917 (Radon 1917) sentó las bases de la reconstrucción 3D a partir de proyecciones[118].

El teorema de Radon afirma que una función real en un espacio euclídeo n-dimensional que obedece a ciertas condiciones de regularidad está definida unívocamente por sus integrales a lo largo de todos los hiperplanos de dimensión n-1. Esto quiere decir que una función definida en tres dimensiones (un espécimen biológico) puede ser reconstruida a partir de sus proyecciones bidimensionales (2D).

El uso de las computadoras para resolver el problema de la reconstrucción añade la complejidad de trabajar en un espacio discreto (ver *funciones base* explicadas en la subsección 2.5.5) en lugar de contínuo.

Por simpilicidad, supongamos que nuestro espécimen es bidimensional. Para reconstruirlo, por tanto, emplearemos una matriz de dos dimensiones $f(x, y)$ que representa a nuestro espécimen. Cada pixel, por motivos de discretización tendrá un valor constante f_j. La dimensión de la matriz es cuadrada, con un total de N pixels, tal que $1 <= j <= N$.

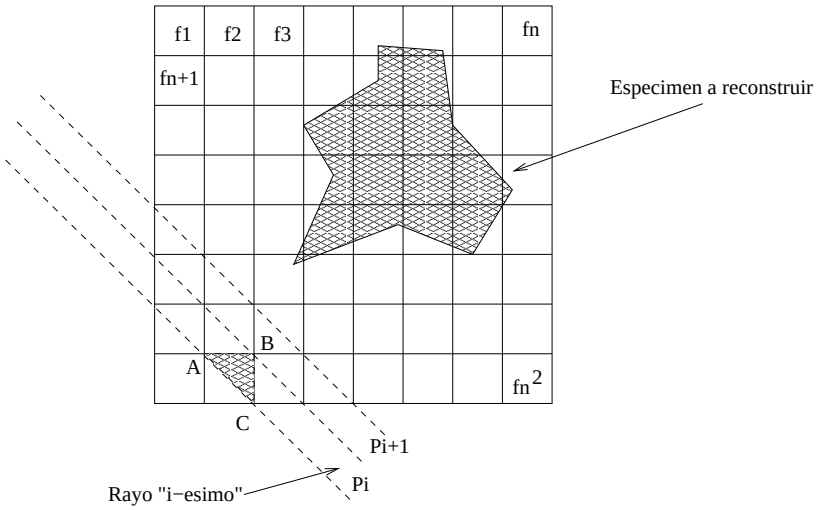


Figura 2.4: Proyección del espécimen 2D.

Cada rayo r_i se envía a través de la superficie en estudio. Un rayo determinado experimenta una interacción con todos los pixeles que encuentra en su trayectoria. Cada interacción se traduce en una intersección entre rayo y pixel, como se muestra en la figura Fig. 2.4. El área marcada entre los vértices ABC se traduce como la contribución de ese pixel al valor final p_i relativo al rayo r_i. Por otro lado cada pixel representa un valor f_j que depende, por ejemplo, de la densidad en ese punto. Así pues la contribución de cada píxel depende de dos factores:

- Área de intesección entre el pixel y el rayo.

- Densidad f_j de cada pixel.

Por tanto cada p_i quedaría definido como se muestra en la siguiente ecuación:

$$\sum_{\substack{1<i<M \\ 1<j<N}} w_{ij} f_i = p_i \tag{2.2}$$

donde M es el número de total de rayos (todas las proyecciones) aplicados al espécimen. w_{ij} es la contribución del pixel j a p_i (dependiente del área de intersección y de la densidad del pixel). El conjunto de todos los p_i de una proyección (todos los rayos paralelos entre sí y en la misma dirección) se denomina *imagen de proyección.*

Las proyecciones serán la fuente de información para los algoritmos de reconstrucción. Un factor vital para la calidad de la reconstrucción es el número de proyecciones disponibles y el número de muestras por proyección. Se generan proyecciones a intervalos iguales en un rango de ángulos, en nuestro caso entre -60 grados y +60 grados.

Las proyecciones se recogen en el llamado una sinograma. El sinograma es una imagen donde cada línea representa la proyección en un ángulo. El nombre de sinograma proviene del la hecho de que cada punto de la imagen original se muestra en el sinograma como una curva sinusoidal. La amplitud de la curva describe la distancia del punto concreto al centro en la imagen original y la fase es el ángulo de proyección.

Este ejemplo 2D se puede extender al caso que nos ocupa, 3D, en este caso la matriz será bidimensional y la función de discretización dejará de ser 2D (pixel) para ser 3D (voxel o blob).

2.3.2. Proyección y retroproyección

El mecanismo de proyección-retroproyección es la base de muchos de los algoritmos de reconstrucción existentes en la actualidad. Supongamos que la matriz que muestra la figura Fig.2.5 representa el mapa de densidad de un espécimen en estudio.

Tras someter al espécimen a la proyección obtendríamos la información mostrada en la figura Fig.2.6(suponiendo que sólo recuperamos dos imágenes de proyección).

El proceso de retroproyección consiste en *repartir* hacia *atrás* las densidades entre todos los pixels (elementos de discretización. La figura Fig.2.7 muestra este proceso. Se puede observar que la reconstrucción genera un *objeto* 2D similar al espécimen 2D bajo estudio, con la diferencia que en el modelo computacional aparece *cierta* distorsión en la matriz. Más adelante veremos

0	0	0	0
0	1	1	0
0	1	1	0
0	0	0	0

Figura 2.5: Espécimen 2D simple sobre el que mostrar el proceso de proyección y retroproyección

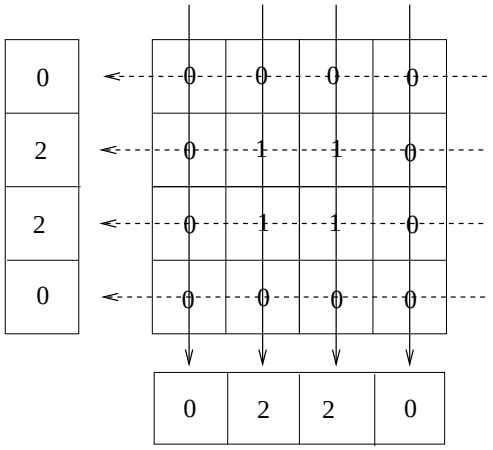


Figura 2.6: Ejemplo proyección. Cada rayo obtiene las densidades acumuladas a su paso. Proceso simplificado.

que este efecto no deseado puede eliminarse.

La figura Fig.2.7 representa a la reconstrucción del espécimen representado en la figura Fig.2.5. Este proceso simplificado pretende mostrar el fundamento empleado por los algoritmos de reconstrucción así como mostrar que el proceso de reconstrucción acarrea consigo defectos que luego se propagan como *ruido* en la reconstrucción. Ruido que se ha de eliminar con técnicas de filtrado.

2.3.3. Teorema de la sección central

El teorema de la sección central [103] establece que la transformada de Fourier (Barrett 2003; Bracewell 1999) de una imagen de proyección 1D de un objeto 2D, es igual a una sección central (valores en una línea que pasa por el centro) de la transformada 2D de Fourier del objeto.

Por lo tanto, cada una de las proyecciones del objeto representa una de estas líneas que

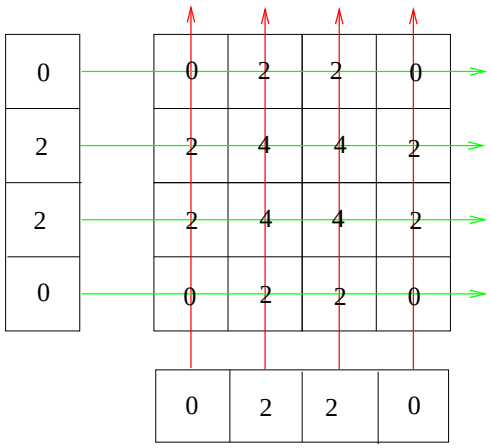


Figura 2.7: Ejemplo retroproyección. Cada reparte las densidades acumuladas a su paso. Proceso simplificado.

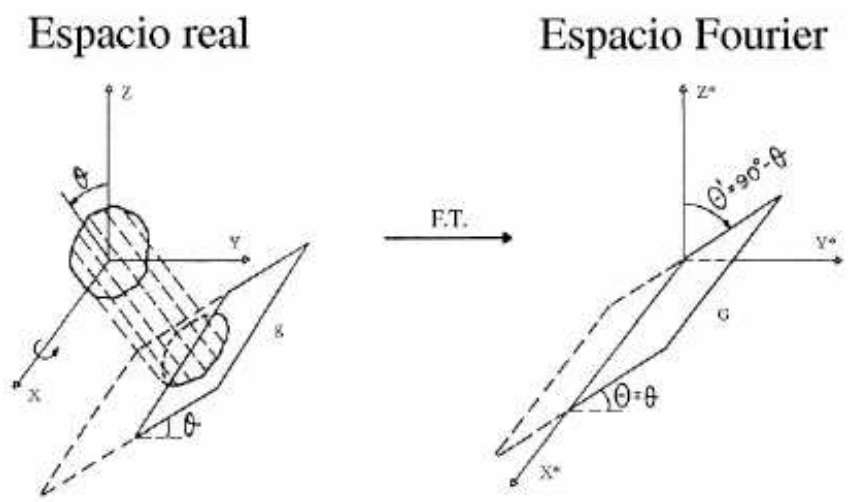


Figura 2.8: Teorema de la sección central.

cruzan el centro de la transformada de Fourier 2D del objeto. El ángulo que estas transformadas forman entre sí es el mismo que existe entre las distintas proyecciones obtenidas en el espacio real. Por lo tanto, una gran cantidad de imágenes de proyección transformadas al espacio de Fourier hacen posible reconstruir la representación en dicho espacio del objeto en estudio. En ese momento, una transformada inversa 2D nos dará como resultado el objeto reconstruido que buscamos.

Este teorema es extensible al caso de reconstrucciones 3D. En ese caso, la transformada del objeto es un volumen y la transformada de la proyección es bidimensional formando entonces un plano del espacio de Fourier centrado en su origen y orientado según esté orientada la proyección

Algoritmo 2.1 : Pseudocodigo del proceso de reconstrucción.

1. Inicializar el volumen
2. **for** $i = 1$ **to** N
3. Para cada imagen I de proyección
4. Proyectar (algoritmicamente) S para obtener I2
5. Calcular el error (I - I2)
6. Relajar la condición de error con el parámetro lambda
7. Retroproyectar el error sobre S
8. **return** Volumen reconstruido;

donde

N Número de elementos de discretización.

I Una imagen de proyección del volumen.

I2 Una imagen de proyección calculada.

lamda Parámetro de relajación.

S Volumen en reconstrucción.

de la que proviene, es decir, es una sección central de la transformada 3D del objeto (ver Fig.2.8).

2.3.4. Expansión en serie

Un método alternativo al *Teorema de la sección central* consiste en establecer una relación matemática entre los valores de los diferentes elementos base que componen el volumen y las proyecciones obtenidas para así obtener la reconstrucción. Estos métodos relacionan proyecciones y elementos base mediante un sistema de ecuaciones lineales.

El tamaño del sistema resultante, el truncamiento de los datos al ser digitalizados y el ruido presente en las proyecciones, hacen inviable la resolución por métodos exactos. Estos inconvenientes hacen de los métodos iterativos de resolución de sistemas de ecuaciones lineales la herramienta ideal.

Los algoritmos iterativos parten de un volumen inicial. Este volumen será refinado en sucesivas iteraciones hasta alcanzar el resultado final. Una iteración finaliza tras haber sido procesadas todas las proyecciones. El volumen se refina progresivamente. El proceso de actualización del volumen depende de la técnica de reconstrucción empleada.

El parámetro λ citado en el paso 6 del proceso de reconstrucción se destina a evitar la divergencia del algoritmo. La elección de este parámetro es clave en el proceso de reconstrucción [118].

La figura Fig.2.9 muestra esquemáticamente el proceso descrito en el breve algoritmo anterior.

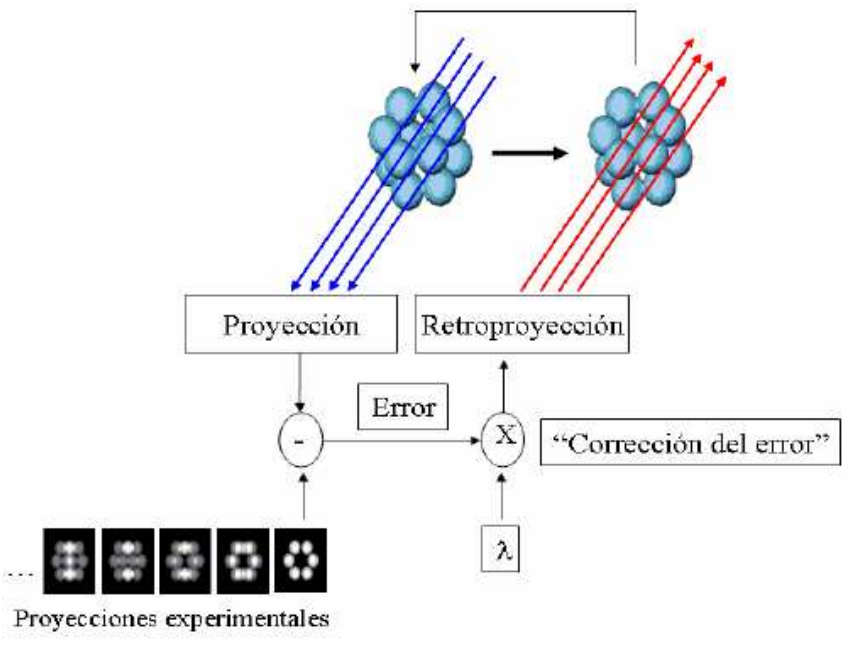


Figura 2.9: Muestra del proceso iterativo de reconstrucción. Cálculo de los errores y aplicación del parámetro lamdba.

2.4. Métodos de Reconstrucción

El método Weighted BackProjection (WBP, en adelante) trabaja de la siguiente forma: Asume que las imágenes de proyección representan la cantidad de densidad de masa encontrada por los rayos de electrones. El método simplemente distribuye la densidad presente en las imágenes de proyección entre cada rayo que se usará para la retroproyección. En este sentido, la masa se retroproyecta en un volumen de reconstrucción. Cuando se repite este proceso para una serie de imágenes de proyección obtenidas desde diferentes ángulos, los rayos de *retroproyección o backprojection* de las diferentes imágenes intersectan y refuerzan en esta intersección aquellos puntos en los que se había masa en la estructura original. Por tanto se lleva a cabo la reconstrucción 3D de la masa del objeto a partir de series de imgágenes de proyección bidimensionales. El proceso de retroproyección tiene el inconveniente de que genera volumenes difuminados. En compensación a la función de transferencia del proceso de retroproyección, ha de aplicarse previamente un filtro paso-alta (weighting) a las imágenes de proyección, de ahí el término *weighted backprojection*. La figura Fig. 2.10 muestra un esbozo del proceso. Para una descripción más

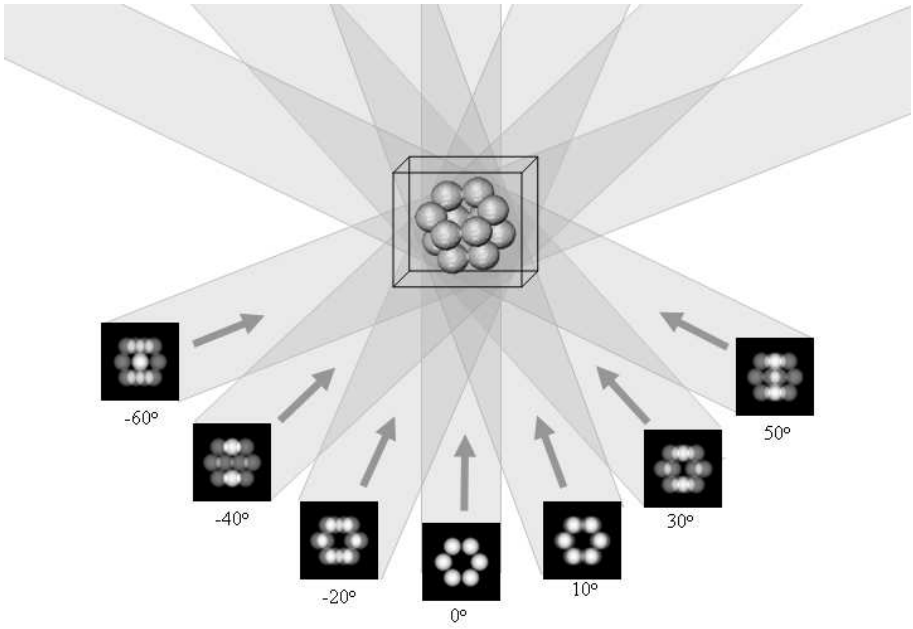


Figura 2.10: Reconstrucción 3D a partir de proyecciones con el método de retroproyección o backprojection. Las imágenes proyetadas se retroproyectan para obtener el volumen.

detallada se recomienda acudir a [119].

La importancia de este método en la tomografía electrónica reside en la simplicidad computacional del mismo (O($N^3 \times M$), donde N^3 es el número de voxels (unidad de discretización elemental del método, equivale a un pixel pero con tres dimensiones) del volumen y M es el número de imágenes de proyección). Los principales inconvenientes del método son que los resultados se ven especialmente afectados por las condiciones de rango angular limitado y baja SNR.

Por otra parte los métodos de expasión en serie se presentan como alternativa a los WBP en la reconstrucción 3D a partir de proyecciones. Se caracterizan por poseer un mecanismo de regularización inherente que los convierten en un método robusto y capaz de afrontar los factores que afectan a WBP [120, 121]. La desventaja de estos métodos es su demanda de recursos computacionales.

La figura Fig. 2.11 ilustra el comportamiento del método WBP y del método de expansión en series SIRT (Simultaneous Iterative Reconstruction Technique) en un caso sencillo, donde no existe ruido. A la izquierda se muestra un volumen artificial que emula a una macromolécula biológica. Se sometió al espécimen a la toma de proyecciones en un rango de ángulos comprendido entre $[-60^o, +60^o]$ en intervalos de 2^o. Se obtuvieron sesenta y una proyecciones. Tomando estas

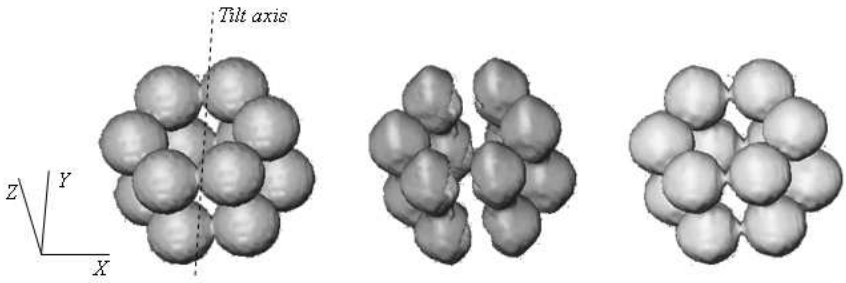


Figura 2.11: Comparación de los métodos de reconstrucción en un caso sencillo.Izquierda: modelo a reconstruir. Centro: reconstrucción con WBP. Derecha: Reconstrucción tras 100 iteraciones de SIRT.

imágenes de proyección se abordó la reconstrucción usando WBP (centro de la figura) y SIRT tras 100 iteraciones (derecha de la figura).

En la figura se puede comprobar cómo afecta a la reconstrucción con WBP la elongación en Z. Efectos que no se aprecian en SIRT. Por otro lado el método iterativo precisó dos órdenes de magnitud de tiempo de computación más que WBP.

Los métodos de expansión en series tienen ciertas ventajas sobre los métodos WBP [122], [120], [123] y [121]:

- Primero, muestran mejor comportamiento en geometrias de adquisición de datos donde el rango angular de toma de muestras está limitado, debido a la como estos métodos completan en rango angular de manera inherente.

- Muestran más robustez bajo condiciones de ruido extremas, dado el proceso de regularización implicito en los métodos iterativos.

- Su flexibilidad en la relación espacial entre el objeto a ser reconstruido y las medidas tomadas, lo que permite usar diferentes funciones base, plantillas para distribuirlas y espaciado entre ellas.

- Finalmente, la posibilidad de emplear restricciones espaciales, que implica que cualquier información *a priori* sobre el objeto o el proceso de formación de la imagen puede usarse para controlar la convergencia hacia la solución.

Los principales inconvenientes de los métodos de expansión de series son :

- Sus demandas computacionales son muy exigentes.
- Se apoyan en la combinación de muchos elementos, como funciones base, modelo de formación de imágen, etc. Todos estos parámetros han de ser optimizados para garantizar una solución factible en tiempo [118, 121].

2.5. Métodos de Reconstrucción Iterativos

Los métodos de expansión en series asumen que el objeto tridimensional o la función f que se desea reconstruir puede aproximarse usando una combinación lineal de un conjunto finito de funciones base conocidas y prestablecidas b_j.

$$f(r, \phi_1, \phi_2) \approx \sum_{j=1}^{J} x_j b_j(r, \phi_1, \phi_2) \tag{2.3}$$

(donde (r, ϕ_1, ϕ_2) son coordenadas esféricas), y que el objetivo es estimar las incógnitas x_j. Estos métodos asumen que las medidas dependen linealmente del objeto de tal forma que :

$$p_i \approx \sum_{j=1}^{J} a_{i,j} x_j \tag{2.4}$$

donde p_i denota la $i$ésima medida de f y $a_{i,j}$ el valor de la $i$ésima proyección de la $j$ésima función base.

Bajo tales presunciones, el problema de la reconstrucción de la imagen puede modelarse como el problema inverso de estimar los x_j a partir de los p_i simplemente resolviendo el sistema de ecuaciones lineales que proporciona la ecuación (2.4). Este sistema de ecuaciones consiste en M ecuaciones, una por cada rayo, con N incógnitas, donde cada incógnita representa a una componente (o función base). Evidentemente la representación del sistema de ecuaciones se traduce en la disposición de matrices que a continuación se presenta.

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1N} \\ a_{21} & a_{22} & \dots & a_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{M1} & a_{M2} & \dots & a_{MN} \end{pmatrix} \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_N \end{bmatrix} = \begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_N \end{bmatrix} \tag{2.5}$$

Este sistema de ecuaciones (Ec. 2.5) describe la relación existente entre la atenuación de

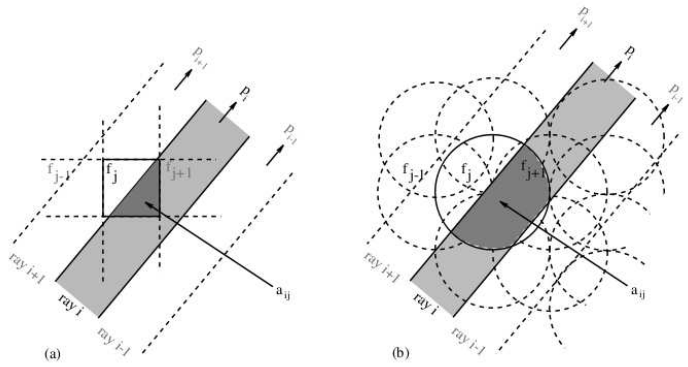


Figura 2.12: Los valores $a_{i,j}$ de la matriz A pueden representar el radio del pixel (a) o del blob (b), f_i que atenúa al rayo i-ésimo. En el caso del pixel, el radio es el área mientras que en el caso del blob se debe integrar la función blob para obtener el radio.

los rayos, las proyecciones y el objeto. Cada fila del sistema de ecuaciones describe la progesión o el avance de un rayo desde el emisor hasta el detector, como cada elemento contiene a una componente, el sistema de ecuaciones indica como afecta el paso del rayo i a cada componente. El término de la izquierda indica la cantidad de atenuación de cada rayo la incidir en cada componente. La mayor parte de estas componentes serán cero pues cada rayo sólo atraviesa una porción de componentes limitada. La suma de todas las atenuaciones es la proyección p_i, cada rayo equivale a una ecuación del sistema.

La incógnita en el sistema de ecuaciones (Ec. 2.5) (si tomamos en consideración la notación matricial Ax=P) es por supuesto la matriz (f_1, f_2,..., f_N), ya que representa a la imagen reconstruida. N es el número de componentes que se usan para describir a la imagen. Las variables conocidas son la matriz P y la matriz A, donde p_1, p_2, p_3, $\cdots$, p_M son los valores de proyección, obtenidos de algún sistema de medición, por ejemplo, rayos-x. A es una matriz M x N que describe cómo cada porción del objeto (cada componente) participa en la atenuación del rayo que atraviesa el mismo. El contenido de la matriz A está fuertemente influenciado por el tipo de función base que se escoja para la reconstrucción. Existen muchas formas de formalizar la relación entre el rayo y las funciones base (componentes) en un número, una de las formas más comunes es la de colocar en cada elemento $a_{i,j}$ de la matriz A el ratio o proporción en la que la componente j-ésima es atravesada por el rayo i-ésimo, como se puede apreciar en la figura Fig. 2.12

Tal sistema de ecuaciones se resuelve, normalmente, con métodos iterativos. Las técnicas

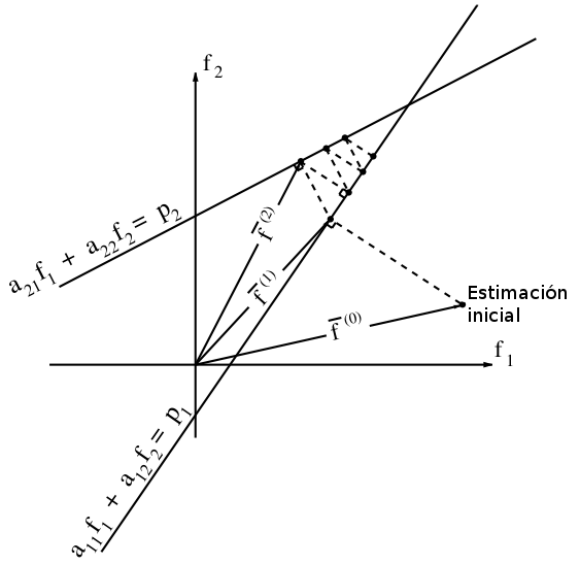


Figura 2.13: Ejemplo del proceso iterativo usando las ecuaciones mostradas.

de reconstrucción algebráica -ART (Algebraic Reconstruction Techniques)- constituyen una de las técnicas iterativas más conocidas empleadas para resolver tales sistemas [118]. Ya que el tamaño del sistema y la falta de una solución exacta hacen que los métodos analíticos sean poco apropiados. Aplicar un método iterativo significa que a partir de una estimación inicial, la solución se busca procesando ecuación tras ecuación. El orden en el que se escogen las ecuaciones afecta a la velocidad y a corrección del algoritmo. Es en este aspecto donde difieren las diferentes aproximaciones iterativas que existen.

La representación de la imagen reconstruida (f_1, f_2, f_3, $\cdots$, f_N) puede verse como un punto en un espacio N-dimensional. Cada una de las M ecuaciones pueden interpretarse como hiperplanos. Cuando únicamente existe una solución para el sistema, es porque existe un único punto donde todos estos hiperplanos intersectan. Este punto es la solución para el sistema. Este concepto se ilustra en la figura Fig. 2.13

La estimación inicial representada en este ejemplo por $\overline{f}^0$, puede ser cualquier punto del espacio N-dimensional en el que se esté trabajando, se proyecta sobre la primera ecuación generando la estimación $\overline{f}^1$ que a su vez es proyectada sobre segunda ecuación obteniendo otra estimación, $\overline{f}^2$, y así se itera proyectando sobre las ecuaciones y acercándonos a la solución que siempre será encontrada, si ésta es única. Matemáticamente podemos expresar las proyecciones sobre los hiperplanos con la siguiente expresión:

$$\overline{f}^{k+1} = \overline{f}^{k} + \frac{\overline{f}^{k} \cdot \overline{a}_i - p_i}{\overline{a}_i \cdot \overline{a}_i} \cdot \overline{a}_i \tag{2.6}$$

Donde $\overline{a}_i = a_{i1}, a_{i2}, \cdots, a_{iN}$ y $\overline{a}_i\overline{a}_i$ es el producto escalar de $\overline{a}_i$ por sí mismo. La fórmula de la proyección se implementa calculando la diferencia entre p_i que es la proyección medida y $\overline{f}^{k+1} \cdot \overline{a}_i$ que es la proyección calculada. Estas proyecciones se encuentran en el sinograma experimental y en el calculado respectivamente.

Todos los algoritmos algebraicos se construyen con la idea de que los sinogramas calculados y experimentales pueden ser idénticos. En otras palabras, la proyección calculada de la imagen reconstruida puede ser idéntica a la proyección medida del objeto real. La fórmula de la proyección determina la diferencia entre estas dos proyecciones y actualiza la imagen reconstruida de modo que la diferencia entre ambas es minimizada. Esta actualización no tiene por qué hacerse de inmediato. Incluso hay diferentes formas de actualizar las diferencias lo que da origen a diferentes algoritmos iterativos de reconstrucción. La ecuación anterior (Ec. 2.6) es por tanto la base de todos los algoritmos algebraicos. Todos usan este método de proyecciones para acercarse iteración tras iteración a la solución. Como se ha dicho antes, la solución será encontrada si es única, lo que no tiene por qué ser cierto en un sistema sobrederterminado M > N. En tal caso el algoritmo ha de acercarse lo más posible al lugar donde se acota la solución. El sistema sobre determinado se va a dar en la mayoría de los casos pues este significa que estamos trabajando con un número de rayos M superior al de componentes (N) y la resolución de la imagen será mejor. El peor caso sería un sistema infra determinado donde M < N. Es decir, el caso preferido es aquel donde contamos con más rayos que componentes. Esto supondrá tener matrices más grandes y más dispersas, lo que complica la computación del problema.

Otro factor a tener en consideración en estos algoritmos iterativos y algebraicos es que cuentan con un factor de relajación λ, no exactamente necesario desde el punto de vista matemático pero si muy útil desde la perspectiva de la reconstrucción. Este factor $\lambda = 1$ no tiene efecto sobre el algoritmo. Pero si se le asigna un valor diferente de 1, actúa sobre la velocidad del proceso de reconstrucción. Un λ muy alto significa dar pasos muy grandes hacia la solución (lo que puede convertir el algoritmo en inestable) y al contrario, un λ pequeño significa añadir cautela al proceso.

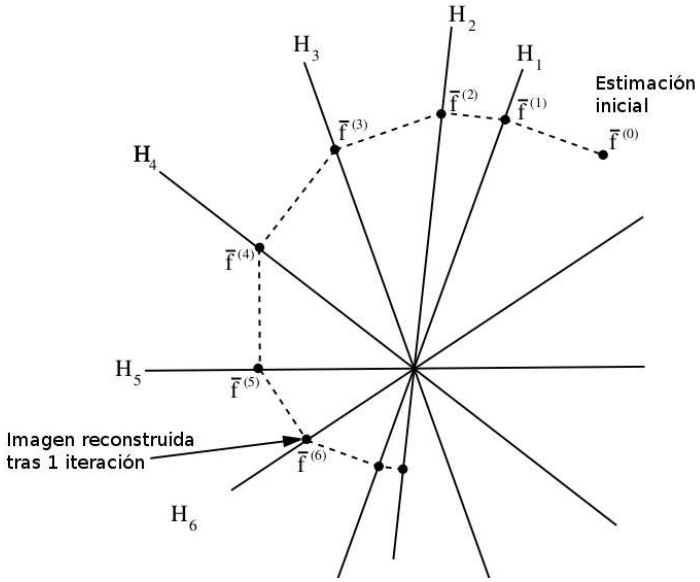


Figura 2.14: Progreso del algoritmo ART, las líneas H_1 a H_6 simbolizan ecuaciones y cada punto es la solución actualizada el proceso iterativo.

2.5.1. ART : Algebraic Reconstruction Technique

Es el más sencillo de los algoritmos de reconstrucción. Proyecta la actual en cada hiperplano (ecuación) una tras otra (ver Fig. 2.14). En ART una proyección en un hiperplano H_i se puede describir de la siguiente forma:

$$\overline{f}^{k+1} = \overline{f}^{k} + \lambda \cdot \frac{\overline{f}^{k} \cdot \overline{a}_i - p_i}{\overline{a}_i \cdot \overline{a}_i} \cdot \overline{a}_i \tag{2.7}$$

Cuando todas las ecuaciones han sido tratadas, se ha dado fin a una iteración del algoritmo. Una solución aceptable necesita varias iteraciones.

2.5.2. SIRT: Simultaneous Iterative Reconstruction Technique

SIRT adopta un sentido totalmente diferente al que tiene ART. En lugar de actualizar la imagen tras cada proyección, SIRT proyecta la estimación actual $\overline{f}_k$ en todos los hiperplanos antes de actualizar (ver Fig. 2.15). Tras esto, se calcula la media de todos los puntos proyectados y éste será la nueva estimación $\overline{f}_{k+1}$. Cuando la media se ha calculado, entonces la suma de todas las correcciones se divide entre el número de rayos (M). Tal y como muestra la ecuación:

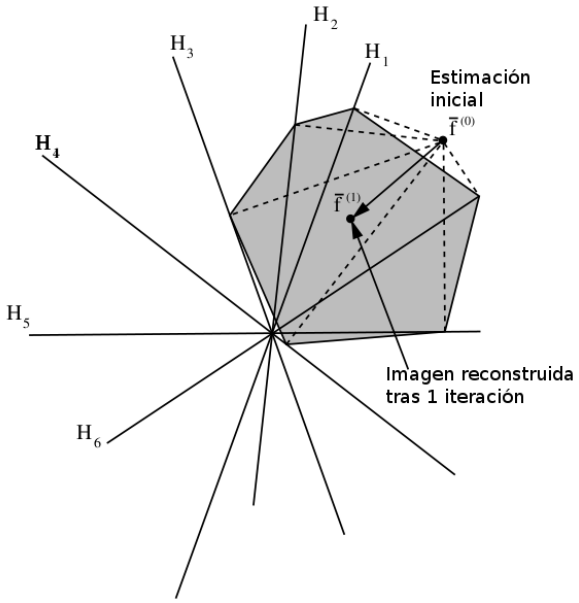


Figura 2.15: Progreso del algoritmo SIRT, las líneas H_1 a H_6 simbolizan ecuaciones y cada punto es la solución actualizada el proceso iterativo.

$$\overline{f}^{k+1} = \overline{f}^{k} + \frac{\lambda}{M} \cdot \frac{\overline{f}^{k} \cdot \overline{a}_i - p_i}{\overline{a}_i \cdot \overline{a}_i} \cdot \overline{a}_i \tag{2.8}$$

SIRT, debido a la media, avanza más lentamente hacia la solución, no obstante esto le hace ser un algoritmo más estable.

2.5.3. SART: Simultaneous Algebraic Reconstruction Technique

Es un algoritmo que está metodológicamente entre ART y SIRT. Trata de combinar la velocidad de ART y la estabilidad de SIRT. Lo que consigue adoptando la metodología de SIRT ya que divide los hiperplanos en grupos y en cada grupo opera como SIRT. Y entre grupos, opera como ART. Matemáticamente SART puede expresarse :

$$\overline{f}^{k+1} = \overline{f}^{k} + \frac{\lambda}{\sum_{i \epsilon G_l}(1)} \cdot \sum_{i \epsilon G_l} \frac{\overline{f}^{k} \cdot \overline{a}_i - p_i}{\overline{a}_i \cdot \overline{a}_i} \cdot \overline{a}_i \tag{2.9}$$

Donde G_l es el grupo que contiene a todos los miembros del grupo l. La suma de las correcciones se divide entre los miembros del grupo. Este algoritmo depende de cómo uno divida las ecuaciones en grupos. Si todas las ecuaciones pertenecen al mismo grupo, entonces estamos

trabajando con SIRT. Si cada ecuación es un grupo, entonces estamos trabajando con ART.

El gran defecto de estos métodos es que consideran la media entre todos los rayos, contribuyan a cada componente o no, el siguiente método CAV se fundamenta en resolver esto, ignorando a todas las componentes que sean cero.

2.5.4. CAV : Component Averaging Methods

El método CAV (Component Averaging Methods) ha mostrado ser [116, 117] eficiente en la resolución de grandes sistemas de ecuaciones lineales dispersos. En esencia, estos métodos han derivado de los métodos ART [118], con la importante innovación de compensar el sistema en función de la dispersión del mismo. Esta compensación dependiente de cada componente proporciona al método una tasa de convergencia que puede llegar a ser superior a la de los métodos ART, especialmente en las iteraciones iniciales. Si se supone que el total de las ecuaciones que conforman el sistema lineal (Eq. (2.4)) pueden subdividirse en B bloques, cada uno con S ecuaciones, entonces se puede expresar una versión general de los métodos CAV a través de la expresión iterativa entre la estimación k-ésima de la solución a la estimación $(k+1)$-ésima:

$$\overline{x}_j^{k+1} = \overline{f}_j^k + \lambda_k \sum_{s=1}^{S} \frac{y_i - \sum_{v=1}^{J} a_{i,v}\overline{f}_v^k}{\sum_{v=1}^{J} s_v^b (a_{i,v})^2} a_{i,j} \tag{2.10}$$

donde:

λ_k denota el parámetro de relajación.

$b = (k \bmod B)$, muestra el índice del bloque.

$i = bS + s$, representa la i-ésima ecuación de todo el sistema.

s_v^b denota el número de veces que la componente $\overline{f}_v$ del volumen contribuye con valores no nulos a la ecuación en el bloque b-ésimo.

El procesamiento de todas las ecuaciones en uno de los bloques genera una nueva estimación de la solución. Todos los bloques se procesan en cada iteración del algoritmo. Esta técnica produce iteraciones que convergen hacia una solución ponderada de mínimos cuadrados del sistema de ecuaciones, siempre que los parámetros de relajación se encuentren en determinados rangos y que el sistema sea consistente [117].

La eficiencia de los métodos CAV se deriva de la dispersión del sistema, representada por el término s_v^b en la Eq. (2.10).

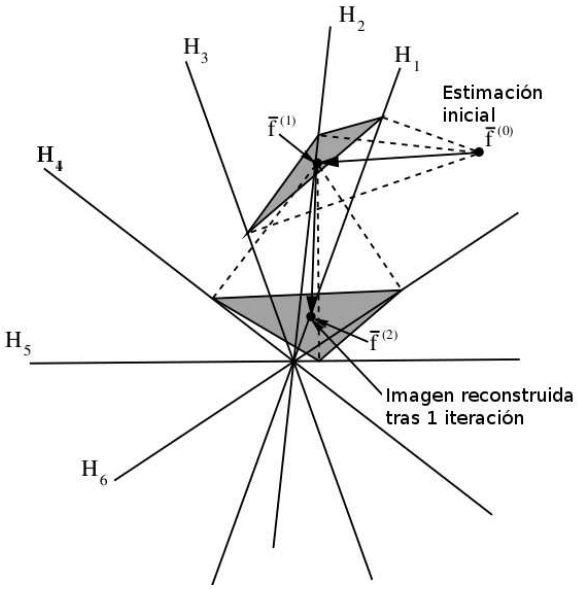


Figura 2.16: Progreso del algoritmo CAV.

Esta ponderación basada en componentes hace que los métodos CAV progresen a través de las iteraciones basadas en proyecciones oblícuas en hiperplanos que constitutyen el sistema lineal [116, 117], ver Fig 2.16. Los métodos ART tradicionales, por otra parte, se basan en proyecciones ortogonales (en ART, $s_v^b = 1$). Las proyecciones oblícuas permiten que estos métodos tengan una convergencia mayor, especialmente en las primeras etapas.

Los métodos CAV, al igual que los ART, pueden clasificarse en las siguientes categorías en función del número de bloques involucrados por iteración:

Secuencial : Este método opera ecuación a ecuación generando estimaciones consecutivas ($S = 1$). Este método coincide con el conocido método ART denominado *row-action* [118], se caracteriza por converger rápido si el factor de relajación es apropiado.

Simultáneo : Este método se conoce, denominado simplemente "component averaging" (CAV) [116], usa sólo un bloque ($B = 1$), considerando todas las ecuaciones del sistema en cada iteración. Este método es inherentemente paralelo, cada ecuación puede procesarse independientemente del resto. Se caracterizan por una tasa de convergencia lenta. Sin embargo, CAV posee una tasa de convergencia inicial, bastante superior a la de los métodos ART simultáneos (SIRT, por ejemplo).

Iterativos por bloques : Block-Iterative Component Averaging methods (BICAV) representan

el caso general. En esencia estos métodos iteran secuencialmente bloque-a-bloque, y cada bloque se procesa de manera simultánea. BICAV muestra una tasa de convergencia superior a CAV y similar a la mostrada por los métodos ART (row-action), siempre que el tamaño del bloque y el parámetro de relajación estén bien seleccionados. Los métodos basados en BICAV (con $S > 1$) son paralelizables.

El trabajo de esta tesis se centra en el algoritmo BICAV (block iterative Component AVeraging).

2.5.5. Funciones Base

En el campo de la reconstrucción de imágenes, la elección de la función base para representar al objeto (digitalizar) que se desea reconstruir influye decisivamente en el resultado del algortimo [124]. Los *blobs*, elementos de volumen esféricamente simétricos con suaves transiciones a cero, han sido investigados [124, 125] en profundidad como una alternativa efectiva al uso de *voxels* en el ámbito de la representación de imagenes. Todos los estudios arrojan como conclusión que los blobs, dadas sus propiedades, son más apropiados para representar estructuras naturales de todos los tamaños. El uso de blobs como funciones base, además, provee al algoritmo de reconstrucción de un mecanismo de autoregulación implícito. En este sentido, los blobs son especialmente indicados para trabajar en situaciones donde las condiciones del ruido puedan ser un incoveniente, generando reconstrucciones mucho más suaves donde tanto el ruido como otras impurezas son reducidas sin mermar la resolución. Especialmente en la tomografía de electrones, el potencial de los algoritmos iterativos de reconstrucción que emplean funciones base esféricas, respecto a WBP han sido puestas de manifiesto usando una metodología objetiva [120, 121].

Los blobs son una generalización de las funciones ventana usadas en el procesado digital de señales denominadas funciones de *Kaiser-Bessel* [124]. Los blobs están limitados espacialmente y, en práctica, tambien puede considerarse que están limitados en banda. La forma del blob y sus características especiales se controlan a través de tres parámetros:

- *radio*
- *el orden de diferenciabilidad*
- *la caída de la densidad*

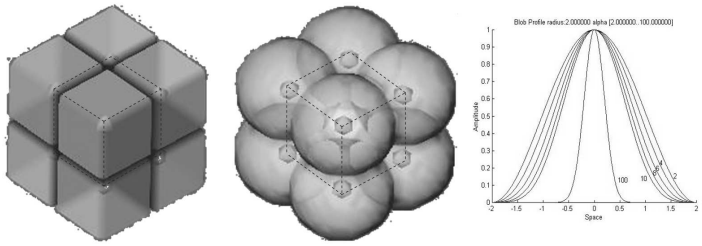


Figura 2.17: Blobs. Derecha: perfil de un blob: función continua con suave transición a cero. Centro: Disposición de blobs en una malla cúbica. Izquierda: Para facilitar la comparación, disposición de los voxels en la misma malla cúbica que se usa para el caso de blobs.

La elección de estos tres parámetros es de extrema importancia. Existe un compromiso entre la resolución deseada y la supresión del ruido. Un blob estrecho genera reconstrucciones de mucha resolución. Un blob muy ancho facilitan la supresión del ruido, para una descripción más detallada se aconseja consultar [124].

Las funciones base, como se puede ver en la ecuación (2.3), son versiones desplazadas de la configuración de blob elegida, emplazándolos en una malla cúbica, que es la misma malla empleada para emplazar las funciones de discretización cuando se usan voxels, con la importante diferencia de que los blobs se solapan entre ellos. Dada esta naturaleza solapante, la disposición de los blobs cubre todo el espacio con una pseudo-continua distribución de densidad (ver Figs. 2.17 y 2.1) muy apropiada para la representación de imágenes. Por otro lado, los voxels modelan la estructura con cubos no solapantes incapaces de proporcional la función de continuidad propia de los blobs, generando discontinuidades espaciales a la hora de implementar la reconstrucción.

El uso de blobs dota de rendimiento a las etapas de proyección y retroproyección de cualquier método iterativo, ya sea ART o CAV. La simetría esférica de los blobs hace que la proyección delo blob sea la misma en cualquier dirección. Consecuentemente es posible crear proyecciones pre-computadas y disponerla en una tabla para su uso posterior [125]. De esta forma el cálculo de los términos $l_{i,j}$ (Eq. (2.10)) en las etapas de proyección y retro proyección se transforman en simples referencias a la tabla. Es más, la aproximación de la reconstrucción empleando blobs hacen factible el uso de algoritmos incrementales. Como consecuencia, el uso de blobs como funciones base para la reconstrucción nos permite obtener mejoras sustanciales en recursos computacionales en comparación con el uso de voxels [125].

2.6. Estrategia de paralelización

Durante años la computación paralela se ha considerado como el vehículo idóneo para facilitar el acceso, al campo de la computación de altas prestaciones, a aplicaciones caracterizadas por ser de gran escala y por suponer un gran reto en escenarios convencionales de ejecución. En el campo de la tomografía electrónica, concretamente la tomografía de grandes especímenes, la computación paralela ya se ha aplicado para portar aplicaciones de reconstrucción de gran índole a arquitecturas paralelas, tanto en el caso de los métodos WBP como en el caso de los métodos iterativos donde la función de discretización es el voxel [110].

Los algoritmos de reconstrucción aplicados a los casos en los que las muestras han sido tomadas con una geometría de adquisición basada en un único eje de giro, ofrecen una paralelización relativamente directa para las arquitecturas masivamente paralelas, siempre que las funciones base que se usen sean los voxels. La reconstrucción de cada uno de los slices es responsabilidad de cada uno de los nodos de la máquina paralela, siempre que los slices tengan un grosos igual al ancho de un voxel y además sean ortogonales al eje de giro. En este caso estaríamos ante un ejemplo de reorganización *impresionantemente paralela* -caso que se da cuando los datos pueden procesarse en paralelo, sin dependencias- del problema de la reconstrucción (vease [126] para poder repasar los conceptos considerados clave en la computación paralela) que no se podría aplicar si usamos las metodologías de reconstrucción basadas en blobs, tal y como se va a demostrar. En su lugar, si se usa el blob como función base, la descomposición de datos necesaria para abordar el problema de manera eficiente requiere de un mayor esfuerzo. Nuestro grupo de investigación cuenta con miembros dedicados a la paralelización y mejora de los métodos de reconstrucción ([121], [127], [128], [129], etc.)

2.6.1. Descomposición del problema

Las geometrías de adquisición de datos normalmente empleadas en la tomografía de electrones, permiten aplicar el modelo de computación paralela *Single-Program Multiple-Data* (SPMD). En este modelo de computación, todos los nodos en la computadora paralela ejecutan el mismo código (el mismo programa), pero para diferentes datos (se realizan particiones del dominio de datos del problema y se asignan a cada nodo). Las geometrías de un único eje de giro permiten que esta división del dominio del problema se haga de manera que se puedan seleccionar slices ortogonales al eje de giro y se agrupen formando conjuntos de slices (slabs).

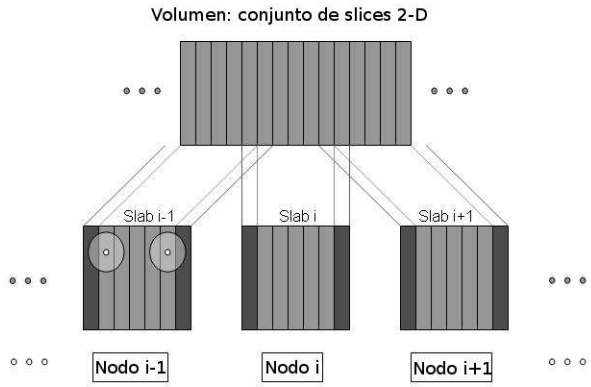


Figura 2.18: Descomposición del volumen: Los slices ortogonales al eje de giro se dividen en conjuntos (slabs) de slices. El número de slabs es igual al número de nodos. Cada nodo de la máquina paralela recibe un slab. El slab contiene un conjunto de slices que son propios (unique, caracterizados con un color gris claro) y slices redundantes adicionales (gris oscuro) dependiendo de la extensión del blob.

En el modelo SPMD empleado para esta descomposición, el número de slabs iguala al número de nodos en la máquina. Cada nodo reconstruye su propio slab.

Si la función base empleada en la reconstrucción es el voxel, entonces cada nodo podría trabajar independientemente sobre su propio conjunto de datos (slab). No obstante, el usar blobs cuyo radio es mayor que la unidad, hace que estos sean blobs que se solapan (los blobs de un slice abarcan parte del slice vecino) de modo que los slices están necesariamente relacionados unos con otros dada la naturaleza solapante de los blobs y además los slabs vecinos están relacionados porque sus slices límite comparten funciones base. La figura Fig. 2.1) puede ilustrar perfectamente este efecto. Por tanto, cada nodo de la computadora paralela recibe un slab compuesto por su subdominio de datos correspondiente y además contiene slices redundantes de los nodos vecinos. El número de slices redundantes que van a ser compartidos depende del radio que se haya seleccionado para el blob. La figura Fig. 2.18 muestra el esquema de esta división de los datos.

Los slices del slab que recibe un nodo determinado se clasifican en las siquientes categorías (en la figura Fig. 2.19 se presenta un esquema gráfico de los diferentes tipos):

Halo Estos slices son únicamente necesarios para poder reconstruir algunos de los slices que

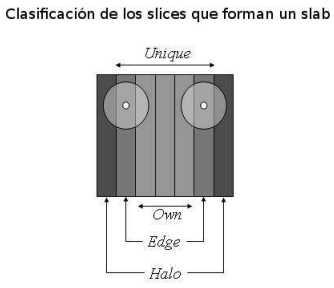


Figura 2.19: Descomposición del volumen: slices replicados y slices propios, los replicados son halo (gris oscuro) y son copia de los nodos vecinos. Los propios se reconstruyen sin necesidad de slices externos (propios) o necesitan información de los halo para poder reconstruirse (edge).

le han sido asignados dentro del slab y que son slices de cuya reconstrucción, ese nodo, es responsable. Son slices, por tanto, redundantes y se disponen en los extremos del slab (como se aprecia en la figura Fig. 2.19). Los slices Halo provienen de los nodos vecinos, estos slices se reconstruyen en los nodos vecinos.

Unique Estos slices se reconstruyen por el nodo. En el proceso de reconstrucción, se usa la información que proviene de los nodos vecinos (debido al citado solape que introduce el trabajar con blobs de radio mayor que la unidad). Estos slices que son propios del nodo, se subdividen en dos categorías:

Edge Slices que necesitan la información *de los slices del vecino*, es decir, los Halo. Estos son slices que se generan en el nodo vecino y que se copian en el nodo actual para permitir la reconstrucción.

Own Son los Slices que no requieren información de los nodos vecinos (de sus slabs). En consecuencia estos slices son independientes de los slices de nodos vecinos, no usan a los slices Halo.

Es evidente que los slices edge de un slab son los slices halo en el nodo vecino.

2.6.2. El método iterativo y paralelo de reconstrucción

Una de las paralelizaciones más comunes a realizar en los algoritmos iterativos de reconstrucción, tanto la versión iterativa por bloques (block iterative) (con $S > 1$, es decir con varias

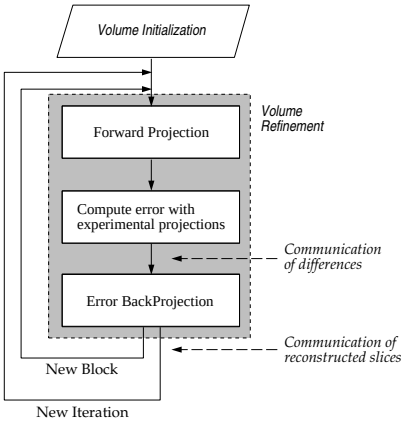


Figura 2.20: Metodo iterativo de reconstrucción paralelizado. Diagrama de flujo del algoritmo donde se representan las etapas en las que se divide.

ecuaciones por bloque) como la versión simultánea de los métodos CAV (Component AVeraging), es la que se implementa siguiendo el modelo SPMD junto con la descomposición de datos descrita. La versión ART (row-action) de estos métodos ha sido obviada ya que el método BICAV (block iterative component averaging) muestra una convergencia similar pero con speedups mejores dada su naturaleza paralela (aunque existen trabajos en el seno del grupo de investigación en los que se ha desarrollado la paralelización de ART trabajando a un nivel mucho más bajo que el propuesto aqui, a un nivel sub-imagen, para mayor información sobre la implementación y los resultados consultese [129]).

Conceptualmente, los algoritmos de reconstrucción que nos ocupan (block-iterative y simultáneo) han de descomponerse en tres etapas consecutivas:

1. Computación de la proyección aplicada al volumen modelo.

2. Cálculo del error existente entre la proyección calculada y la proyección experimental.

3. Refinamiento del volumen modelo a través de la propagación hacia atrás o retro-proyección del error calculado.

Estas etapas pueden ser identificadas claramente en la ecuación (2.10). Los algoritmos deben abordar cada etapa para cada bloque de ecuaciones y para cada iteración, tal y como se muestra en el diagrama de flujo representado en la figura Fig. 2.20. El volumen modelo inicial ha de ser reiniciado a cero, a cualquier valor constante o bien al resultado de una reconstrucción anterior.

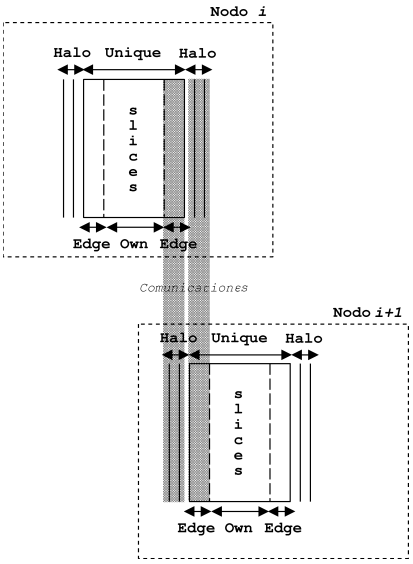


Figura 2.21: Comunicaciones en el algoritmo paralelo. En cada punto de comunicación, cada nodo actualiza la información de sus slices Halo con la información de los slices edge de cada nodo vecino. Antes del paso de la retroproyección del error la información que se intercambia es la del error calculado, tras el paso de la retroproyección del error, la información que se intercambia es la de los slices reconstruidos.

La relación de dependencia entre slices vecinos a causa de la extensión del blob implica que, para poder computar tanto la proyección del volumen como la propagación hacia atrás (o retro proyección) del error para refinarlo, de un determinado slab, es necesario que existan fases de comunicación entre nodos vecinos. Especificamente, las comunicaciones se refierene al intercambio necesario de los slices Halo para posibilitar la reconstrucción de los slices edge. El diagrama de la figura Fig. 2.20 muestra un esquema del algoritmo iterativo, indicando los puntos donde se va a realizar la comunicación, estos son antes y despues de la propagación hacia atrás del error calculado. La figura Fig. 2.21 muestra un esquema donde se describen cuales son los slices involucrados en las comunicaciones: los slices halo se actualizan con la información de los slices edge del nodo vecino.

La aproximación SPMD permite, tras el intercambio de información, el procesamiento en paralelo e independiente de cada slab por cada nodo de la arquitectura paralela. Obviamente este procesamiento depende de dos puntos de sincronización implícitos que son justamente los puntos de comunicación indicados anteriormente.

En cualquier desarrollo que involucre comunicación entre nodos de una red, aparece la necesidad de *ocultar las latencias*. Ocultar las latencias hace referencia a la práctica de solapar las

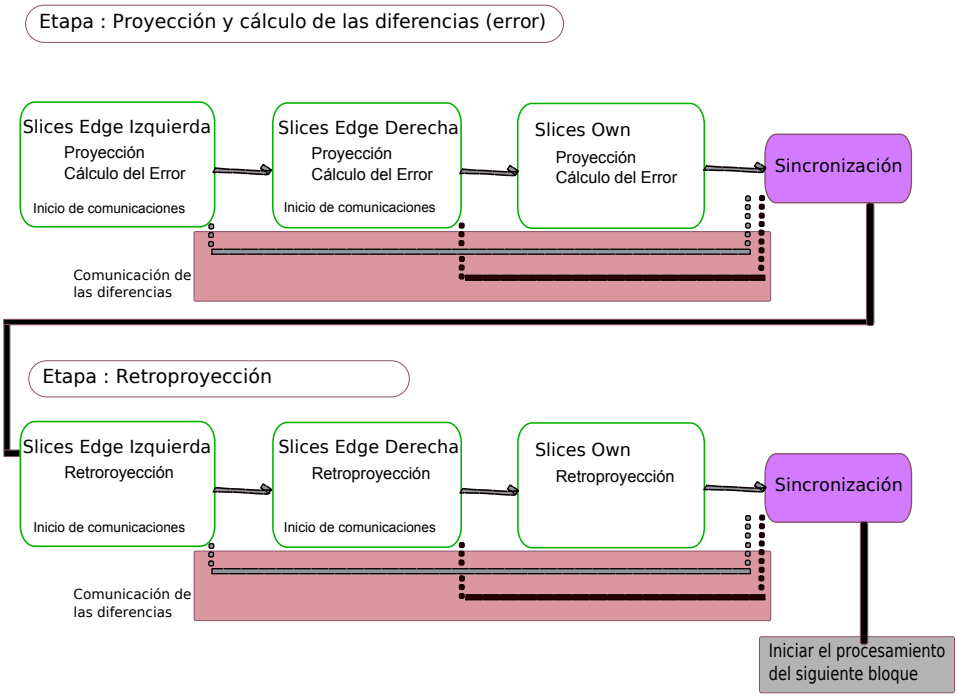


Figura 2.22: Técnica de ocultación de latencia.

etapas de comunicación de un nodo (momento en el que la CPU queda inactiva y a la espera de que finalice la etapa para proseguir) con computación útil para el nodo en un futuro cercano (ya se citó en el capítulo uno de esta tesis la abstracción *futures* que se encarga precisamente de esto). El modelo de trabajo implementado en los algoritmos a paralelizar invita a aplicar una técnica de ocultación de latencia consistente en alterar el orden en el que se procesan los slices de un slab [121]. La figura Figura 2.22 esquematiza la operación de ocultación de latencia implementada en [121]. El proceso de ocultación de las latencias consiste en procesar los slices edfe situados a la izquierda del nodo, de modo que puedan ser enviados a los nodos vecinos cuanto antes (para que éstos consigan sus slices Halo y puedan procesar los slices Own). Mientras se envían estos slices y durante esta operación de comunicación, el nodo aborda la tarea de procesar los slices edge de la derecha. Mientras se remiten estos slices, el nodo solapa la comunicación con el procesamiento de los slices Own. Esta operación se aplica en los dos puntos de comunicaciones, antes y despues de la etapa de envío del error. Además, el modo en el que se orienta la comunicación entre los nodos hace que esta implementación paralela este exenta de interbloqueos.

Por último, la descomposición de datos vista en la subsección anterior hace que el problema sea balanceado pues a todos los nodos se les asigna el mismo número de slabs (y posiblemente

con el mismo tamaño). Esta aproximación puede fallar en estar balanceada si el sistema paralelo en el que se ejecuta la aplicación paralela no es un sistema de uso excluyente, es decir, pueden existir más de una ejecución a la vez en la máquina paralela. A esto se le debe añadir el caso en el que los slabs no quepan en memoria y cada nodo deba ir a disco a por él.

2.6.3. Implantación de la reconstrucción en la plataforma cluster

La computación en los Cluster [130] ha sido y es una alternativa efectiva en coste para acceder a la supercomputación ya que se apoya en el uso de hardware convencional y de componentes software estandar. Una de las características que además hacen destacar a los cluster sobre los supercomputadores es el turn-around time (tiempo que pasa desde que el programa se ha lanzado hasta que los resultados están disponibles) que en los clusters es mucho más bajo. Esto se debe a los tiempos de espera que nuestro programa debe soportar en los sitemas de colas de estas máquinas.

La disponibilidad de APIS como MPI (Message-Passing Interface) [131] permite a los programadores implementar aplicaciones paralelas de manera independiente a la arquitectura para la que se desarrolla. Lo que hace que la mayoría de los científicos y programadores prefieran desarrollar sus aplciaciones paralelas usando C estándar y MPI.

Los resultados de las paralelizaciones siguiendo esta descomposición de datos y esquema de desarrollo, pueden encontrarse en el trabajo [121].

2.6.4. Algortimos de difusión como modelo de estudio : Jacobi3D

El paralelismo de datos ocurre en algoritmos donde se pueden aplicar el mismo número de operaciones a cualquier partición de los datos, allí donde estén los datos, de manera independiente y en paralelo. Este tipo de algoritmos son también conocidos como *embarazosamente paralelos o impresionantemente paralelos* como se vió en la sección 2.6. Otra cuestión diferente es el *paralelismo de tareas* donde intervienen factores como la comunicación entre procesos (o hebras, según el caso), dependencias de datos, etc. En nuestro, paralelismo de tareas, hay situaciones en las que hemos de estudiar el patrón de comunicaciones exclusivamente. En el desarrollo de esta tesis hemos usado una implementación paralela particular de un *algoritmo de difusión*. Un algoritmo de difusión es un algortimo iterativo que se aproxima a la solución iteración tras iteración (de manera similar al esquema de la reconstrucción usando métodos iterativos). En

cada iteración se usa la solución de la iteración anterior. En nuestro caso empleamos (Jacobi) en matrices de tres dimensiones, y sigue el mismo patrón de comunicaciones que el problema de la reconstrucción. La difusión de Jacobi es un algoritmo iterativo que aproxima las ecuaciones diferenciales de Laplace (se pueden consultar los detalles de este método en el capítulo 19.5 del libro *Numerical Recipes in C* [132]).

Capítulo 3

Reconstrucción en Entornos Multihebrados. Ocultacion de Latencia

3.1. Introducción

Las aplicaciones propias del área de ciencia e ingeniería, suelen implementar complejos modelos que simulan o abstraen el fenomeno motivo de estudio. Un gran porcentaje de estas aplicaciones adopta un esquema o *patrón de ejecución* iterativo [133] donde cada iteración implica el procesamiento de estructuras de datos, a menudo gestionadas por intrincadas organizaciones de punteros. Durante todo el capítulo se ha tenido en cuenta el aforismo de **Nicklaus Wirth** (*programa = algoritmo + estructuras de datos*) otorgándole el peso adecuado a las estructuras de datos. Los datos con los que trabajan estas aplicaciones distan de apoyarse en estructuras regulares y en la mayoría de los casos provocan iteraciones heterogéneas en cuanto a intensidad de cómputo se refiere. No obstante, se espera de este tipo de aplicaciones el comportamiento dinámico citado ya que refleja el fenómeno simulado. Los requerimientos computacionales demandados por estas aplicaciones hacen necesario *el uso de la computación paralela o supercomputación.*

Las nuevas arquitecturas que están apareciendo en escena tratan de superar los límites experimentados por la última generación de procesadores [134], amortiguando la tesis impuesta por la regla de Pollack [16] por la que el rendimiento de cada procesador está ligado a la complejidad interna del mismo. Estas últimas propuestas computacionales, se apoyan en construcciones en las que la escala de integración juega un papel importante [135], como es el caso de las arquitecturas multicore y de las GPUs (Graphics Processing Units) que pueden suponer una continuación de

la ley de *Amdahl* según autores como Xian-He Sun [136]. Tal es así que algunos autores incluso proponen abandonar el modelo de computación usado en las arquitecturas paralelas como los clusters [137].

Si se torna la vista hacia atrás, se puede ver que las arquitecturas que hoy abanderan el cambio no son otra cosa que la evolución natural de una arquitectura nacida a principios de la década de los 70, los *transputers* [138]. Tras un cómodo periodo en el que se ha podido extraer rendimiento sin más que factorizar las arquitecturas, la industria se ha visto obligada a retomar el modelo que desde hace décadas ha venido rechazando. Los motivos de que esta arquitectura no tuviera un éxito merecido puede encontrarse en que para usar estas arquitecturas eficientemente, se precisaban conocimientos sobre concurrencia y paralelismo (cualquiera que haya usado *Occam* puede hacerse a la idea de lo tedioso que pudo resultar aquello). Por lo que ante la decisión de continuar programando con el modelo tradicional (programación estructurada liderada por un único flujo de control) o dar el paso al entorno paralelo−concurrente impuesto por estas arquitecturas, ocurrió lo que tenía que ocurrir y es que se relegó esta última opción a favor de la primera. Los avances tecnológicos no pueden seguir siendo explotados, como hasta ahora, para seguir creando cada vez procesadores más y más potentes. Por lo que hemos de volver hacia atrás. Hoy día, por suerte, los modelos de programación son más completos [29], las técnicas de depuración están más conseguidas [49], programar en estos entornos hoy, no es tan malo como antes. Quizás este paréntesis haya sido lo que necesitábamos para aprender a programar tal y como hoy se requiere que lo hagamos o para estar dispuestos a adoptar el cambio que proponen trabajos como [134].

Es evidente, por todo lo anterior, que el modelo de supercomputación que la comunidad científica ha estado explotando durante las dos últimas décadas está experimentando cambio. Tradicionalmente se han usado librerías de paso de mensaje para adaptar aplicaciones propias del ámbito de la ingeniería y la ciencia, a las arquitecturas paralelas. Librerías como MPI [139] se han convertido casi en un estandar de facto en este aspecto. Estas aplicaciones, desarrolladas haciendo uso de MPI, carecen del soporte dinámico tan característico de ellas. En [140] se pueden encontrar argumentos en este sentido. La eficiencia de estas aplicaciones, en este caso, no se ve explotada en toda su potencia, en este sentido se pueden encontrar trabajos como [141] y [142] que detectan esta rigidez impuesta por las librerías de paso de mensajes y se esfuerzan por incluir mecanismos que flexibilicen los desarrollos. A nuestro juicio estos mecanismos fallan al

contemplar únicamente al algoritmo y obviar a las estructuras de datos. Sin embargo existen implementaciones como [143] que proponen alternativas a las implementaciones tradicionales, *reescribiendo* toda la librería sobre un modelo de objetos concurrentes [144]. Es este tipo de implementaciones, que contemplan modificaciones a los algoritmos y a las estructuras de datos sobre las que se trabajará durante todo este capítulo para permitir que la computación sobre las plataformas cluster sea más flexible y cómoda consiguiendo también que la portación de las aplicaciones ya desarrolladas sea menos costosa que la reescritura de toda la aplicación.

Las aplicaciones paralelas (y las secuenciales también, por supuesto) que deseen explotar las capacidades de los citados entornos de computación precisan considerar esta dualidad *paralelismo y concurrencia* o por el contrario ser completamente reescritas de modo que el paralelismo pueda ser expresado considerando las condiciones de concurrencia. En este escenario tan cambiante, las **abstracciones** que se empleen pueden ser de mucha utilidad a la hora de mantener las cotas de rendimiento. A lo largo de este capítulo se mostrará cómo las abstracciones empleadas para describir el problema, juegan un rol importantísimo desde la perspectiva de la escalabilidad y rendimiento.

A lo largo de este capítulo examinamos la forma de portar aplicaciones ya desarrolladas a un entorno que permita dotarlas de la flexibilidad necesaria facilitando la labor del programador. Primero se presenta a la aplicación sobre la que trabajamos, su descomposición y dependencia de datos para ser paralelizada en MPI. Luego se presenta un marco de trabajo desarrollado en Java que sienta las bases de lo que buscamos como entorno para dotar de flexibilidad a nuestros desarrollos. Tras esto veremos que existe un marco de trabajo llamado Adaptive MPI (AMPI) [143], que es una extension a la implementación particular de MPI donde el modelo tradicional de proceso se convierte en objeto y se embebe en hebras de usuario.

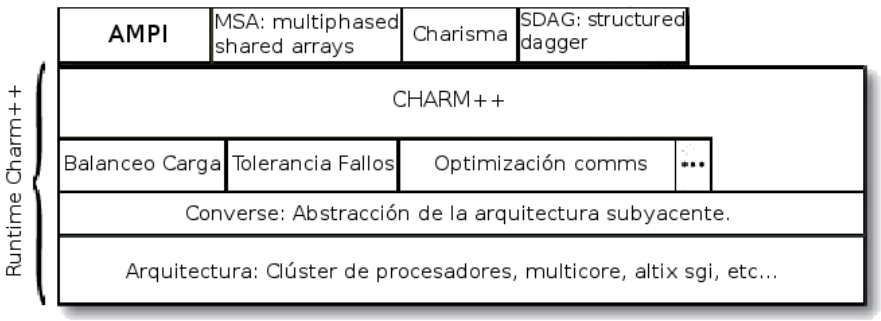


Figura 3.1: Arquitectura del Runtime Charm

Este marco de trabajo permitirá dar luz a la versión de *grano grueso* del problema de la reconstrucción. Se presenta también una segunda alternativa, basada en una implementación del problema usando programación orientada a objetos, esta opción de *grano fino* se construye usando un modelo particular de objetos ofrecido por el marco de trabajo Charm++ [144]. AMPI y Charm++ están relacionados a través del *runtime* (ver figura 3.1) del que forman parte. AMPI es un *framework* de muy alto nivel y Charm++ es otro *framework* que constituye un sustrato de AMPI. En la figura 3.2 se puede ver cómo el *runtime* actúa como capa de abstracción de la arquitectura subyacente y provee de utiles funciones de gestión y planificación.

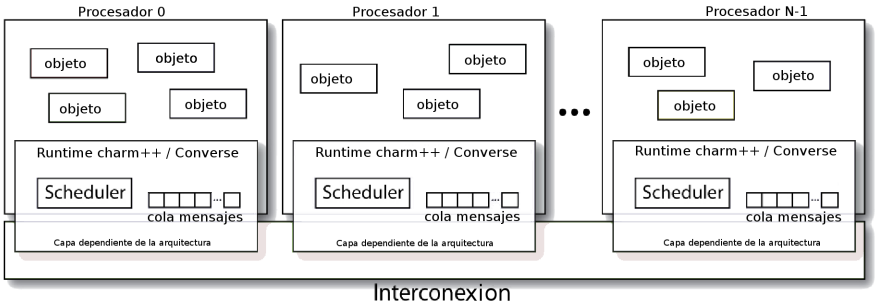


Figura 3.2: Uso del Runtime

La nueva generación de aplicaciones paralelas que se presentan en el campo de la ciencia e ingeniería cambia radicalmente respecto a las que hasta ahora se han venido desarrollando. Esta generación de aplicaciones han de flexibilizarse para poder explotar las arquitecturas sobre las que se ejecutan y disfrutar de un dinamismo que se debe ver reflejado en tiempo de ejecución. En el caso de las aplicaciones de ciencia e ingeniería tradicionales y ya existentes, el dinamismo ímplicito en ellas se ve lastrado por el modelo de programación que además impide que puedan adaptarase con eficiencia a nuevas arquitecturas. El escenario es aún más complejo si la aplicación es de naturaleza irregular [145]. El modelo de programación más usado para implementar programas paralelos en plataformas paralelas es MPI. Este modelo no ofrece el dinamismo necesario del que deben disfrutar las citadas aplicaciones. La Interfaz de Paso de Mensajes (o MPI, segun sus siglas en inglés) se ha convertido en un estandar. De la especificación de tal estandar se han generado numerosas implementaciones. De estas, unas pocas son excelentes como MPICH [146] o MPI-LAM [147], [148] donde se describe el uso de *componentes*. Otra alternativa o implementación de esta interfaz, que ha levantado muchas expectativas en el mundo de la computación de altas prestaciones es Open MPI [149], ya que es un esfuerzo

para llevar al mundo del *open source* la implementación MPI. Las implementaciones de MPI *particulares* tambien existen, estas suelen ser proporcionadas por los grandes fabricantes de arquitecturas paralelas. La mayoría de las implementaciones de MPI son eficientes y están muy optimizadas en el *paso de mensajes*, tengase en cuenta que las comunicaciones es uno de los puntos fuertes del estándar MPI. A pesar de esto, todas las implementaciones de MPI (en un porcentaje muy elevado) no consiguen dotar de la naturaleza dinámica o flexibilidad necesaria a las aplicaciones científicas [140].

Las abstracciones han mostrado ser una herramienta efectiva [150], [145], [151], [152] para abordar el espacio conceptual que separa la concepción del problema según humanos y computadoras. La vertiginosa evolución experimentada por los procesadores, principalmente en su arquitectura, dificulta al software aprovechar todo el potencial que el hardware ofrece tras las sucesivas mejoras de este último. Es en este punto donde las abstracciones juegan un papel importante.

Los clusters de procesadores (computadores) constituyen la plataforma de HPC más rentable y conocida hasta la aparición de las arquitecturas multicore. Las aplicaciones que precisan de una ingente cantidad de recursos computacionales para resolver un problema han hecho un uso extensivo e intensivo de estas arquitecturas. Portar estas aplicaciones a la plataforma cluster es una tarea que reviste cierta dificultad, ya asumida. Llevar una aplicación científica o de simulación a una arquitectura como la citada requiere que se apliquen ciertas técnicas de programación paralela. En nuestro caso particular, el problema de *la reconstrucción de imágenes tridimensionales a partir de proyecciones* puede ser resuelto con solvencia usando técnicas de paralelización avanzadas en la plataforma cluster [121] y [127]. En este caso, y al igual que ha ocurrido con un altísimo porcentaje de aplicaciones, la comunidad científica ha adoptado un modelo computacional basado en *procesos monolíticos* que se comunican a través del envío de mensajes entre procesadores. El modelo usado ha arrojado ciertos beneficios de rendimiento aunque siempre a expensas del programador que ha realizado la implementación (ocultación de latencias, llamadas no bloqueantes, etc). Recientemente están realizando su aparición en escena diversas arquitecturas (procesadores multicore). Quizá, la falta de estandarización existente hasta el momento [153], en estas arquitecturas sea un justificante de lo que se pretende demostrar en este capítulo. Para estas arquitecturas, tan recientes, donde se dispone de más de un núcleo de procesamiento por chip, se deben examinar varios modelos de programación con la intención

de abandonar el tradicional esquema basado en procesos convencionales. Es preciso tener en cuenta que al existir un número mayor de núcleos de procesamiento por chip, la frecuencia del reloj puede reducirse (porque se está en disposición de afirmar que al tener más de un núcleo, se pueden realizar más tareas sin la necesidad de incrementar la frecuencia de reloj) de este modo, uno de los pilares del *power wall* puede ser ignorado [19]. La escala de integración, otro de los problemas asociados con los procesadores tradicionales, se convierte en el caso de los multicore en un gran aliado, pues permite integrar más núcleos por chip. Como deciamos, una aplicación construida sobre el modelo de proceso tal y como se desarrollaban las aplicaciones para las plataformas cluster, experimenta una considerable e inmediata pérdida de rendimiento cuando se usa una plataforma multicore. La explicación es simple, un proceso convencional en un procesador multicore usará un único core, dejando libres el resto. Este núcleo trabaja a menor frecuencia que el procesador donde la aplicación solía ejecutarse. Si la aplicación se diseña para ejecutar un proceso por núcleo, entonces entrarán en juego pérdidas de rendimiento asociadas a ciclos de cpu perdidos durante etapas de comunicación entre procesos. De modo que una aplicación desarrollada bajo esas premisas no podrá experimentar (cuando se use un multicore) las ganancias en rendimiento y escalabilidad esperadas.

Los procesadores multicore son arquitecturas paralelas con memoria compartida. Las técnicas de programación que se pueden usar en ellos se apoyan en el uso de hebras [154] para poder así mantener a todos los núcleos ocupados. No obstante el uso de hebras tampoco es garante de ganancias en el rendimiento, pues implica aplicar ciertas técnicas en aras de respetar la coexistencia de varios flujos de control en el mismo espacio de direcciones [155], [156]. De este modo, y según se afirma en [151], se hace preciso el uso de *abstracciones* para respetar ciertas cotas en las ganancias de rendimiento. En este sentido las abstracciones basadas en el *paradigma de orientación a objetos* ofrecen un método flexible para describir la forma en la que se desarrollará la computación. El modelo en el que se apoya este paradigma es inherentemente paralelo [157], es precisamente esta característica el principal puntal que usaremos para desarrollar todo el capítulo. La granularidad de nuestras implementaciones depende, en cierto modo, de este concepto. Durante el presente capítulo se mostrará cómo usando objetos embebidos en hebras de usuario es posible llevar a cabo ejecuciones de programas exigentes, desde el punto de vista de los recursos que precisan, en plataformas destinadas al HPC, ya sean clustersde computadores o procesadores multicore.

La idea de crear un elemento de procesamiento compuesto por varias entidades subordinadas, como es el caso de los procesadores multicore, no es una novedad, de hecho, un par de décadas atrás se comenzó una tendencia similar, en este caso las arquitecturas usadas fueron los *transputers* [138]. Una compañía denominada *INMOS* dió luz a una CPU modesta pero muy barata que disponía de su propio sistema de comunicaciones *on-chip*. Inteconectando estas CPUs se podía crear un sistema paralelo escalable. Desafortunadamente la citada compañía desapareció por dos grandes razones, la primera por tardar demasiado en liberar la última versión de su procesador y la segunda por la complejidad de la programación que estos *procesadores transputers* necesitaban. Los programadores no estaban dispuestos a pagar el coste de usar un modelo de programación nuevo que *aunaba paralelismo y concurrencia*. El lenguaje usado era *Occam* [158] así que la complejidad era real. *Occam* es un lenguaje concurrente construido sobre la bases del *álgebra de procesos* denominada CSP (Communicating Sequential Processes) [159]. Este álgebra ha sido añadida a lenguajes como Java cuando fueron ampliados para adoptar extensiones para la concurrencia, adaptadas al nuevo lenguaje CTJ (Communicating Threads for Java) [160] y [161]. Las arquitecturas basadas en *transputers* no sólo ofrecieron un punto de vista interesante para manejar la concurrencia en lenguajes orientados a objetos convencionales. Además crearon modelos útiles para el paralelismo, donde ambos conceptos (computación paralela y concurrencia) pudieran ser combinados. Se comenzaba a gestar una atractiva alternativa para expresar el paralelismo a la vez que se tiene en cuenta las condiciones de concurrencia. Las abstracciones basadas en orientación a objetos [162], [163] se usaron para aligerar el sinnúmero de conocimientos y destrezas técnicas requeridos a la hora de desarrollar código paralelo en el que se pudiera explotar la concurrencia (un transputer podía gestionar hebras, una para las comunicaciones y otra para la computación, las tareas paralelas tenían por tanto, una componente concurrente para impedir pérdidas de rendimiento).

Los modelos de programación que proponemos en esta tesis para trabajar tanto con clusterscomo con arquitecturas recientes como los procesadores multicore, y las basadas en GPUs, consideran los avances iniciados con las arquitecturas transputers [164], [157], [165], [162] y [163] aunque los modelos que usamos han evolucionado enormemente de aquellos, las bases conceptuales se heredan en gran proporción de aquellos, ya que la similitud entre transputers y multicores / GPUS es muy alta.

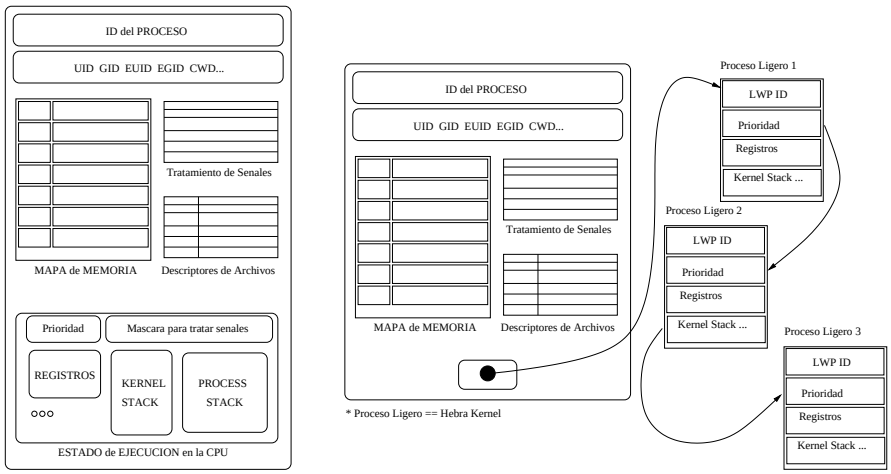
En este capítulo, se recoge la idea de reusar patrones de programación en arquitecturas emi-

nentemente paralelas. Además se presentan resultados sobre aplicaciones paralelas desarrolladas al amparo de estos modelos. En el apartado 3.3 se define el entorno sobre el que basamos los desarrollos. En el apartado 3.4 se presenta un estudio de rendimiento del entorno de desarrollo. En el apartado 3.5 se introducen los conceptos relativos al problema de la reconstrucción iterativa que más nos afecta. La primera propuesta alternativa a los desarrollos actuales se considera en el apartado 3.5.1 donde se estudia una implementación de grano grueso en la que se usa un modelo desatendido y de alto nivel para traducir la entidad de ejecución *proceso* en una más flexible resultante de combinar hebras de usuario y objetos creados por el propio entorno. En el apartado 3.6 se muestra cómo se comporta una implementación de grano mucho más fino, tanto en clusters como en multicores. Esta aproximación se creó a partir de la reescritura completa del código usando orientación a objetos explícita y el framework Charm++ [144]. En el apartado 3.7 se expone el trabajo abierto en relación con este capítulo. Finalmente, el apartado 3.8 resume los aspectos destacables de este capítulo.

3.2. Objetivos

En este trabajo de tesis se ha creado un entorno de trabajo en el que las aplicaciones de ciencia e ingeniería puedan ser dotadas de medios para intentar superar los obstáculos que las diferentes arquitecturas pueden suponer en las ganancias de rendimiento. Se ha abordado el trabajo para dos arquitecturas paralelas muy diferentes, cluster de estaciones de trabajo y arquitecturas multicore. Por un lado se ha establecido una metodología de trabajo para optimizar aplicaciones en clusters(adaptando las aplicaciones ya desarrolladas para estas arquitecturas) y por otro se ha estudiado cómo combinar concurrencia y paralelismo en las arquitecturas multicore. En una segunda parte se han desarrollado estrategias de adaptabilidad para que estas aplicaciones se ejecuten en el mapa óptimo de procesadores que la máquina paralela pueda ofrecer.

Una de las características más complejas de abordar cuando se trabaja con MPI es tratar de ocultar a la aplicación el efecto de la latencia de la red cuando se llevan a cabo comunicaciones. En un escenario ideal, cuando el programa paralelo en un procesador concreto ejecuta una sentencia *receive*, el mensaje que contiene la información se encuentra ya en el procesador que la solicita. Por tanto la detención de la CPU y el gasto de ciclos de CPU necesarios para esperar a la llegada de la información, se reducen a cero. Para conseguir un escenario cercano al ideal, el programador debe idear una estrategia de emplazamientos de *sends* y *receives*, de tal forma que



(a) Estructura clásica de un proceso.

(b) Procesos Ligeros (LWP)

Figura 3.3: Estructura de un proceso clásico frente a un proceso hábil para usar hebras kernel.

entre ellos se pueda emplazar computación, independiente de esa comunicación, que se puede llevar a cabo mientras esta tiene lugar. También se puede optar por usar hebras, característica proporcionada por el estándar MPI−2 [166], pero hay varias razones por las que desaconsejar tal técnica. La primera de ellas es que MPI−2 no es un estándar *en toda su extensión* y el uso de características específicas puede ocasionar incompatibilidades. Otra es que se ha de reescribir todo el código para hacerlo hebrado (y esta opción no está menos exenta de errores que la de disponer de sends y receives no bloqueantes). Otra razón es que a pesar de las dos anteriores, la ejecución del proceso paralelo y de su etapa de comunicaciones se volverían completamente no−determinista y esto se debe evitar [137]. No cabe duda que la característica multihebra es deseable para afrontar muchos de los problemas propios de MPI, pero se hace necesario acompañar a esta característica de determinismo [22]. La programación multihebrada junto con el modelo de programación orientado a objetos sientan las bases para la creación de entornos de trabajo capaces de proporcionar de manera directa tal facilidad [150], [167],[168] y [169].

Es evidente que se precisa el modelo de objetos para otorgar cierto determinismo ante la concurrencia [150], pero también para explotar el paralelismo inherente al dominio del problema [145], [157], [165]. Respecto al uso de la multihebra, afrontamos la decisión de apoyar nuestro trabajo en la estructura que nos ofrecía el *kernel* o por el contrario, confiar en un *runtime* que implementara hebras manejables desde el espacio del usuario. Usar la estructura clásica de proceso, estructura en la que se apoyan actualmente nuestros desarrollos ha demostrado ser poco

flexible ver figura 3.3(a), en este modelo cualquier *cambio de contexto* es muy costoso ya que el *estado de la computación* se refleja en el contenido de los registros de la computadora (contador de programa,puntero de pila, registros de propósito general), la tabla de páginas de la MMU, el contenido de la memoria, los archivos en disco, ... Cuando se requiere, usando el modelo tradicional de proceso, un cambio de contexto se precisa almacenar todo este estado en un área segura y cargar el estado de computación del nuevo proceso, los pasos podrían entenderse de la siguiente forma :

- Se debe almacenar el contenido de todos los registros que está usando el proceso **P1**
- Se debe recuperar, de la estructura de datos que modela un proceso (figura 3.3(a)), el valor de todos los registros que usa **P2** y cargar estos valores en la CPU.
- La CPU vuelve a ejecutar código de usuario y por tanto a ejecutar instrucciones de **P2**, dandose por realizado el cambio de contexto.

El concepto de *proceso ligero* o *light weight process* depende del sistema operativo en uso, el objetivo de esta estructura es habilitar la multitarea. Un LWP se ejecuta en el espacio de usuario, mapeado sobre una hebra de kernel y comparte el espacio de direcciones y los recursos con otros LWP dentro del mismo proceso. Como se puede observar, un LWP es un concepto similar al de hebra de usuario, de hecho hay sistemas operativos que consideran que son idénticos. La utilidad de los LWP reside en la posibilidad de planificar multiples hebras de usuario en uno o varios LWP, lo que permite diseñar políticas de *scheduling* con mucha facilidad. Un buen trabajo donde poder examinar cómo se puede explotar la concurrencia ofrecida por los LWP es el trabajo de Mühlbacher [170] donde subraya la ventaja de emplear los LWP desde la abstracción que proporciona la orientación a objetos.

Si se observa la figura 3.3(b), cuando dos LWP realizan un cambio de contexto, éste ocurre de manera similar. Pero en este caso, el cambio de contexto puede beneficiarse de la caché (warm caché)[171], ya que muchos de los datos que manejan los LWP son globales y comunes. En el caso de querer conmutar dos hebras de usuario, ver figura 3.4, se sigue el mismo procedimiento:

- Una hebra (**T1**) intenta acceder a un recurso bloqueado por otra hebra -**T2**- y como consecuencia se separa de la ejecución. En este caso se comienza (siempre en código de usuario –recuerdese que se usa una librería o *runtime*–) a ejecutar el planificador (ver figura 3.4).

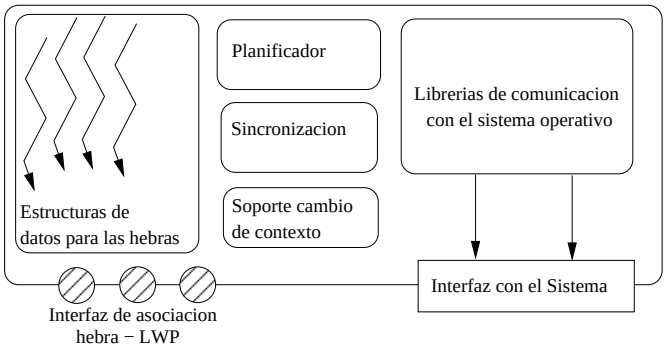


Figura 3.4: Librería de gestión de hebras (User Level Threads)

- La CPU guarda su estado de ejecución en una de las estructuras de datos reservadas para la hebra **T1**, se cargan los valores para otra hebra (**T2** por ejemplo) desde la estructura de datos correspondiente a dicha hebra, estos datos se cargan en los registros de la CPU.

- El planificador termina su ejecución dejando los valores de **T2** en los registros de la CPU. En el caso de las hebras de usuario no se precisa realizar llamadas al Sistema Operativo (el planificador está en el espacio de usuario) por lo que la operación de cambio de contexto es *extremadamente rápida*.

Finalmente adoptamos un marco de trabajo que nos permitiera emplear *hebras de usuario*, por las razones expuestas y porque decidir la politica de planificación a este nivel es mucho más sencillo que desde el kernel del sistema operativo, incluso la *migración* de hebras desde un procesador a otro se simplifica siguiendo este esquema [172].

Por otro lado y una vez analizado el entorno, un gran porcentaje de aplicaciones tienen como característica el poseer una distribución de la carga de trabajo no uniforme. En otro amplio rango de casos, las aplicaciones compiten con otras aplicaciones por usar los recursos del procesador. En la medida que, como primer objetivo, se pretende generar un esquema de trabajo en el que tanto la computación como la comunicación se mantengan en una proporción constante para que el esquema de solapamiento (ocultación de la latencia) sea eficaz, se deben buscar medios para que el peso de la computación sea lo más homogéneo posible. De hecho, de no ser así la estrategia de solapamiento sería efectiva sólo en los procesadores más lentos. La importancia de establecer mecanismos de balanceo, por tanto, es decisiva a la hora de seleccionar a las *hebras de usuario* como las candidatas para nuestro cometido. El balanceo de la carga precisa que las *unidades que embuten a la computación* en nuestras aplicaciones sean capaces de ser *migradas*

sin presentar dificultades. Este, constituye otro motivo para elegir como unidad de trabajo a las *hebras de usuario* además, su instrumentación es fácil de acometer desde el entorno de ejecución, con el fin de ejercer el equilibrado de la carga. Este aspecto se trata con mayor profundidad en el siguiente capítulo.

En definitiva, cualquier aplicación científica desarrollada usando C y MPI (incluso C++) precisa modificaciones en su código fuente con el objetivo de obtener rendimiento en sus ejecuciones. Implementar estas modificaciones supone una tarea tediosa ya que las labores de solape entre comunicación y computación se delegan al buen hacer de quien se encuentra encargado de la implementación. Se apuesta, en todo este trabajo, por el desplazamiento del típico modelo de computación basado en el modelo de proceso a un modelo hebrado en el entorno paralelo y basado en objetos, mucho más conveniente. Este modelo hebrado aprovecha la característica de las hebras de usuario para explotar la concurrencia como un recurso más a sumar en el haber de la aplicación.

Apuntando a las necesidades que describimos, ya se han presentado algunos proyectos en la comunidad científica. Entre ellos cuentan algunas versiones MPI con soporte para programación multihebrada en un procesador de tal modo que se alcanza un grado de *concurrencia adicional* y se explota así el solape entre comunicación y computación. Este es el caso de [173] que usa multithreading para mejorar el rendimiento de programas MPI multihebra en computadores de memoria compartida, o de [141] y [142] que tratan de mejorar las implementaciones de MPI. Sin embargo, el potencial del solape entre comunicación y computación es complejo llegar a que se aproveche en su totalidad pues aún con esta implementación queda en manos del propio programador el tomar decisiones sobre la elección de los flujos de control, mecanismos de planificación, etc.

En definitiva, los dos aspectos principales que buscamos son :

- La capacidad de poder trabajar con un modelo más flexible capaz de aprovechar mejor las capacidades paralelas de las nuevas arquitecturas y por otro que sea eficiente en la tarea de ocultar la lantecia de la red.

- La capacidad de mantener al sistema computacional con una carga de trabajo homogenea (en la mayor medida posible) con la intención de favorecer las tareas de ocultación de latencia.

3.3. Definición del marco de trabajo.

Las aplicaciones con las que trabajamos son aplicaciones desarrolladas para un entorno paralelo tradicional, esto es, aplicaciones desarrolladas en C y MPI. Dotar a estas aplicaciones de la capacidad necesaria para extraer el máximo rendimiento en los escenarios descritos es difícil [140], por un lado es una tarea compleja hacer que estas aplicaciones sean capaces de adaptarse al entorno computacional en el que se ejecutan (la adaptación se debe realizar en la implementación), por otro lado, dotar a estas aplicaciones de la capacidad de ocultar la red de interconexión que emplea cada arquitectura es una tarea que también requiere un gran esfuerzo. La aplicación de trabajo (***BICAV***) ha sido objeto de diversos tipos de actuaciones con el fin de mejorar el rendimiento siguiendo métodos basados en la habilidad del programador, obteniendo buenos resultados [121]. En el capítulo 2 ya se describió el fundamento de esta aplicación. Más adelante en este mismo capítulo podrá encontrar la técnica empleada para paralelizar la aplicación cuando fue programada usando C y MPI. Es intención de este capítulo mostrar cómo podemos mejorar en el rendimiento entre esta versión y la que esta tésis propone obteniendo ventajas complementarias producto de las propiedades del entorno utilizado (adaptabilidad). La nueva versión se apoya en los conceptos defendidos hasta ahora : el uso de hebras de usuario y del paradigma orientado a objetos.

El entorno o marco de trabajo que buscamos, pretende:

- Abstraer, en la medida de lo posible, a la aplicación de la arquitectura que se emplee para ejecutarla. De esta forma la implementación no habrá de ser variada cada vez que se cambie de arquitectura. Este paso se ha de dar sin perjudicar el rendimiento de la misma.

- Debe posibilitar la incorporación de aplicaciones ya desarrolladas. Evitando tener que reescribir toda la aplicación por completo.

- Debe proporcionar un entorno dinámico, en cuanto al emplazamiento de las unidades de computación (procesos, hebras, etc)

Llegados a este punto se plantea la necesidad de decidir entre usar una librería para gestionar las hebras de usuario (similar a la librería *pthreads*[171]) o implementar capas adicionales que además ofrezcan gestión de memoria, planificación, etc. Para nuestro propósito se hace necesaria la segunda opción, especialmente porque buscamos la integración con el modelo de objetos

y porque precisamos la capacidad de adaptar aplicaciones *ya existentes* al nuevo entorno sin necesidad de alterar –agresivamente– la lógica de la aplicación (*como nota aclaratoria tengase en cuenta que el desarrollo de software científico sufre un ciclo de vida complejo y demasiado prolongado en el tiempo, esto hace que cuando la aplicacón está dispuesta a ser usada en el entorno de trabajo, tanto la aplicación, como los desarrollos probablemente hayan quedado anticuados*).

Usar una librería de gestión de hebras significa dejar aún mucho en manos del desarrollador [174], además de que no se facilita la implantación de medios que permitan *portar* aplicaciones con cierta independencia del entorno. En cierto modo, para adoptar la decisión de virar hacia un *runtime* o hacia una *máquina virtual*, valoramos además ciertas aportaciones en [175] y en [169]

En este sentido hace falta una serie de capas *software* que faciliten esta tarea, descargando al programador. En [169] advierten sobre los beneficios de esta aproximación. La decisión sobre hasta qué nivel llegar (*runtime* o *máquina virtual*) ha sido también dificil y decisiva. La diferencia fundamental entre *máquina virtual* y *runtime* es que la máquina virtual abstrae completamente la ejecución de la arquitectura física sobre la que se apoya.

3.3.1. Runtime, mejor opción como entorno de trabajo

AMPI [176] es la capa superior de un entorno (Charm++ [144]) que ofrece a las aplicaciones MPI existentes una interesante opción de virtualización (ver figura 3.1). Igual que en la propuesta analizada (ver apartado 3.7), en AMPI se usa el concepto de *procesador o proceso virtual, se pueden usar ambas acepciones en el contexto de esta tesis* que consiste, grosso modo, en una hebra que *acoge* a un proceso MPI, uno de tantos en los que se divide la computación. La ventaja que proporciona este entorno frente al entorno desarrollado en Java es que no usan una arquitectura simulada (máquina virtual). El inconveniente, por tanto, es que no se puede usar la computación AMPI entre arquitecturas hardware que usen diferentes repertorios de instrucciones (aspecto que pretendemos resolver con la implementación presentada en el apartado 3.7).

En MPI, cada proceso se ejecuta, por tanto, en un procesador individual aunque hay trabajos en los que se estudia cómo hacer coexistir eficientemente a varios procesos en el mismo procesador, como es el caso de [133] a costa de dar soporte a la ejecución a través de un runtime llamado IOS [177]. En contraste con este método y al igual que en el entorno desarrollado (ver

apartado 3.7), AMPI puede ejecutar más de un proceso virtual por procesador gracias al uso de hebras de usuario, en este caso AMPI emplea una implementación de hebras basada en la librería *QuickThreads* [178]. QuickThreads permite implementar librerías de gestión de hebras a medida.

La ventaja que ofrece este runtime (o RTS –Run Time System–) es que, a diferencia de la versión desarrollada, es capaz de *asociar* cada hebra o procesador virtual (en adelante VP –Virtual Processor–) a un procesador físico siguiendo criterios de balanceo de carga (balanceo estático, pues las decisiones se adoptan antes de iniciar la ejecución). Es decir, es el propio RTS el encargado de repartir las hebras entre los procesadores, obteniendo más eficiencia que la versión desarrollada en Java. Además se cuentao con la habilidad de expresar los algoritmos paralelos usando programas que sean independientes del número de procesadores físicos. La expresividad, en esta aproximación se ve favorecida. Esta capacidad no impide que cuando el número de procesadores virtuales sea idéntico al número de procesadores físicos el programa se ejecute de idéntica forma que si la versión MPI fuera la que se encarga de la computación. En el caso de este entorno de ejecución, se consigue amortizar la capa intermedia de software usando un número de procesadores virtuales superior al número de procesadores físicos.

Usar AMPI, que es una de las capas de más alto nivel de todo un complejo sistema ([144] y [179]) proporciona además de la libertad al desarrollador de poder expresar su aplicación con un particionamiento independiente del número de procesadores, la gran ventaja de poder cambiar el entorno de ejecución de la aplicación MPI sin tener que cambiar sensiblemente su código. Como se expresó en el capítulo 1 y tal y como se indica en la literatura [180], además de las ganancias en rendimiento existe la mejora en el proceso de ingeniería del software aplicado por los desarrolladores, visible en términos tangibles como desarrollos con alta cohesión (cada módulo de un programa se entiende como una unidad con significado y cada componente de ese módulo está intimamente relacionado con cualquier otro del mismo módulo) y acoplamiento débil (cada módulo tiene significado por si sólo e interacciona muy poco con otros módulos, lo que beneficia a la reutilización del código, aspecto a considerar).

Un intento de implementar un runtime basado en Java se llevó a cabo en los inicios de este trabajo de tesis (ver apartado 3.7). En este apartado se analizan las ventajas del runtime Charm++ frente al desarrollado por nosotros, indicando el estado en que se encuentra el progreso de desarrollo.

3.3.1.1. Procesos virtuales como abstracción: Solape adaptativo

Uno de los inconvenientes que sufre MPI, es que bloquear la CPU para esperar a que se complete una etapa de comunicación puede ser tremendamente perjudicial para el rendimiento de toda la aplicación. Esta afirmación se aplica sobre todo en las nuevas arquitecturas de supercomputación, equipadas con potentes coprocesadores destinados a las comunicaciones. Coprocesadores que poseen la habilidad de descargar a la CPU de la tarea de comunicar. Razón por la que en estas arquitecturas es imprescindible no bloquear la CPU mientras se procesa la comunicación. Las llamadas no bloqueantes de MPI pueden ser útiles porque son capaces de solapar comunicación y computación (dentro del mismo proceso) la cuestión es si el proceso posee toda la computación necesaria para ocultar las latencias de las comunicaciones. Añadir la capacidad multihebra a MPI (como en el caso de MPI–2) constituye otro esfuerzo orientado a ocultar las comunicaciones, pero esta eficiencia contrasta con el incremento en el esfuerzo que supone trabajar con hebras.

La figura 3.5 muestra un *cronograma* donde se comparan conceptualmente dos escenarios, uno donde sólo se emplea MPI como mecanismo *conector* entre *procesos* paralelos y otro escenario donde en su lugar se emplean los elementos del entorno de trabajo AMPI. En el primer escenario, se comprueba que el módulo A debe *llamar* al módulo B. No existe dependencia entre B y C, el programador ha de decidir el primer módulo a invocar. Hecho esto y durante la ejecución del primer módulo, el segundo módulo queda ocioso, sin posibilidad de poder ser ejecutado. En el escenario presentado sólo se ejecutará el segundo módulo (C en este caso) cuando se haya terminado la ejecución de B.

Este modelo es ineficiente porque no se permite la ejecución del segundo módulo a pesar de no existir dependencia alguna entre B y C. Sin embargo con el entorno AMPI es posible que A *invoque* la ejecución de alguno de los métodos de B y C al mismo tiempo, basta con activar (a través de mensajes) su ejecución (ver figura 3.6 para tener mejor perspectiva de los procesos virtuales o VP). En el segundo escenario la ejecución de los módulos B y C puede *solaparse, entrelazarse* de tal modo que cuando alguno de ellos se bloquee debido a una operación de comunicación o debido a una espera obligada debido a la existencia de desbalanceos de la carga, entonces el otro módulo podrá aprovechar la condición de ociosidad de la *cpu* para avanzar su computación.

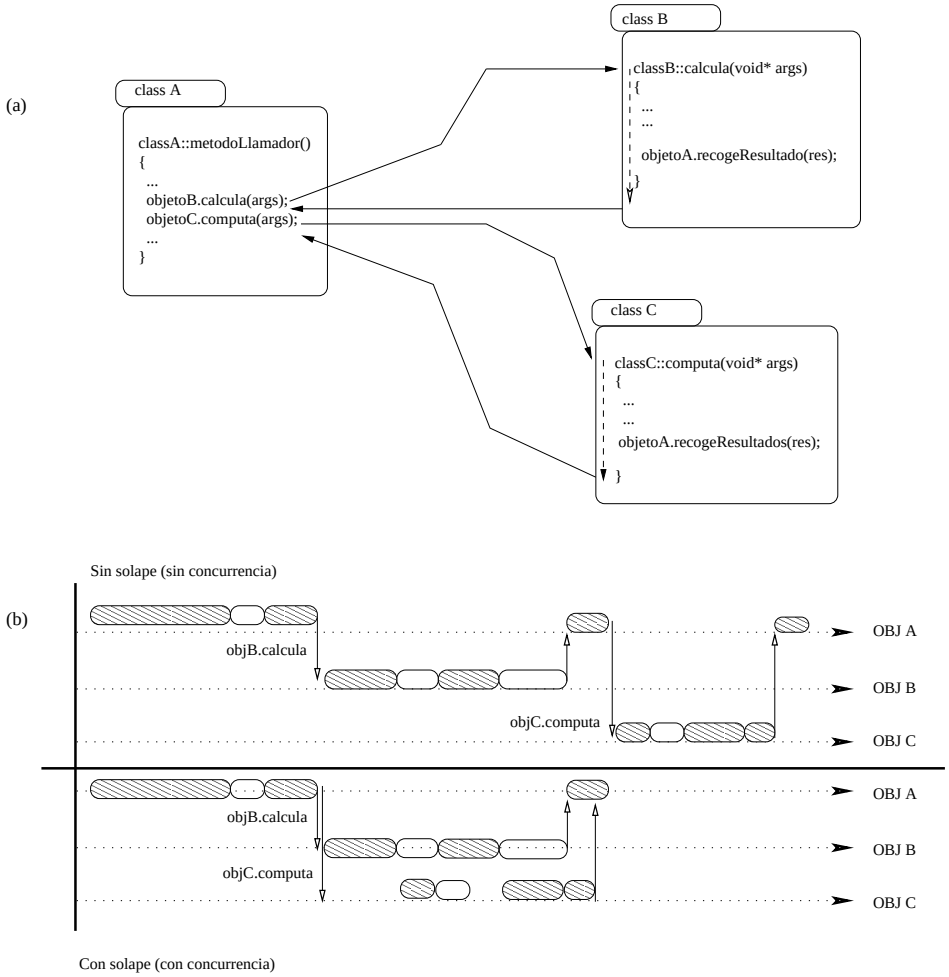


Figura 3.5: (a) Invocación de métodos entre objetos. (b) Representa el *timeline* de la invocación, con solape y sin solape. La ejecución de los métodos de B y C pueden ejecutarse secuencialmente, con lo que entre la llamada al método de B y al método de C existe una diferencia temporal igual a lo que tarda en ejecutarse el método de B. Sin embargo si los objetos son concurrentes, se pueden aprovechar los espacios en los que el procesador queda ocioso durante la ejecución de algún método, para ejecutar sentencias del método de C. De manera que en este caso la llamada al método de B y al método de C se realizan seguidas (salvo dependencias), aprovechando así el potencial solape entre ambos objetos.

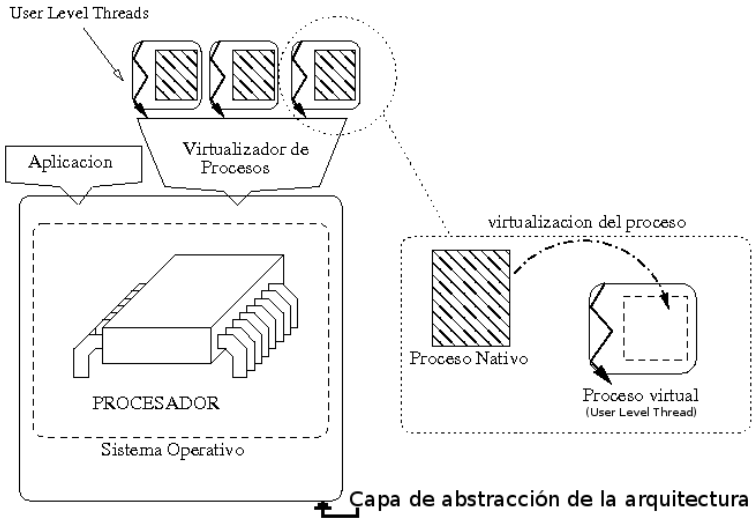


Figura 3.6: Ejemplo de virtualización de procesos a través de las hebras de usuario

Procesos virtuales

El concepto de proceso virtual puede examinarse en la figura 3.6 donde se puede observar que es una *entidad de ejecución* controlada por una hebra. Dentro de esta *entidad* se encuentran todas las instrucciones que se hayan dentro del proceso original que se ejecutaría en un procesador físico en condiciones normales (código C–MPI). Se le denomina proceso virtual porque a diferencia de los procesos reales, estos no tienen una estructura en el *kernel del sistema operativo* (evidentemente esto les otorga mayor flexibilidad e independencia de la arquitectura en relación con sus congéneres, y menos soporte por parte del sistema operativo).

La implementación que AMPI ofrece para materializar este concepto se apoya en *hebras de usuario* donde cada hebra recoge un *objeto paralelo*. El rank que usa un proceso virtual es independiente del rank que caracteriza a un procesador virtual (el rank equivalente a MPI es el que posee el proceso virtual). Hebra y objeto paralelo son uno.

Las hebras empleadas para esta implementación, al igual que para nuestra primera versión del runtime, son hebras de usuario, se crean y se planifican desde código provisto por el usuario (a modo de librería). Las ventajas que se derivan se cuentan entre la posibilidad de ofrecer un tiempo de cambio de contexto extremadamente pequeño, delegan al usuario en control sobre la planificación de cada hebra, sobre el *stack* asignado a cada hebra y la localización del mismo (aspecto importante puest tanto para *cambiar el contexto* como para *migrar* la hebra se

debe acceder a este para poder recoger el estado de ejecución de la hebra). De acuerdo a las recomendaciones de los creadores de este entorno (Parallel Programming Lab de la Universidad de Illinois) existe un *consumo ineludible de tiempo* debido a la gestión de las hebras (cambio de estado entre suspendida, operaciones de planificación, reactivación), del orden una fracción de milisegundo, además las hebras son apropiativas.

Otra característica de AMPI es que la comunicación (paso de mensajes) entre entidades (VPs o procesos virtuales) se realiza a través de mensajes enviados entre los objetos paralelos, mensajes gestionados directamente por el runtime. La gestión de estos mensajes es independiente de la localización física del objeto.

Adaptación y Variables globales

El uso de hebras para embeber procesos impone una serie de condiciones que el código desarrollado con C (y MPI) no contempla. El principal inconveniente encontrado a la hora de portar las aplicaciones al entorno multihebrado lo ofrecen las variables globales. Los programas escritos en C suelen hacer un uso extensivo de las variables globales. Dos hebras –procesadores virtuales– en el mismo procesador físico comparten las mismas variables globales. Modificar una variable global puede dar al traste con la computación asociada a otra hebra que no haya considerado el cambio. La situación puede complicarse aún más cuando entran en juego las variables estáticas locales, que en cierto modo han de contemplarse como variables globales de pleno derecho. Por ejemplo, en el código siguiente se plantea una función compartida por dos hebras, es aquí donde entra en juego el concepto de TSD o Thread Specific Data, pues cada hebra necesita que la variable estática local se comporte (bajo el contexto de la hebra en curso) como tal, sin generar conflictos con las demás hebras.

```
void casoEstaticaLocal(void)
{
  static int local = 1;
  printf("%d\n", local);
  local=2;
}
```

Cuando la primera hebra (T1) ejecute este código por primera vez, el resultado será la impresión del valor 1. La siguiente vez que la primera hebra (T1) ejecute la función, el valor

impreso será 2. Pero, y en contra del significado del valor *static local*, cuando la segunda hebra (T2) invoque a la función por primera vez, debería aparecer el valor 1 y en su lugar aparece el valor 2. De este modo, las variables locales estáticas serán contempladas, a todos los efectos, como variables globales y específicas para cada hebra.

Las soluciones para abordar los inconvenientes relacionados con las variables globales pasan por :

- Crear una copia de las variables globales para cada una de las hebras que participan de la computación. La mejor forma de alcanzar esto es obteniendo una copia del *GOT (Global Object Table* para cada una de las hebras de usuario que participan en la computación. En nuestro trabajo, en ocasiones optamos por esta opción, ya que el runtime proporciona los medios necesarios para que el *scheduler* del sistema pueda albergar copias de la citada tabla por cada hebra y permite activar cada copia cuando su correspondiente hebra está activa. (*La implementación de esta característica en AMPI fue incluída durante una estancia en el grupo de investigación que desarrolla el framework*). Es una técnica que depende mucho del sistema operativo sobre el que se ejecute la aplicación, pues no todos los binarios almacenan esta tabla en la misma posición.

- Eliminar las variables globales del código original haciendolas privadas en el código MPI. Es el método más portable entre sistemas operativos y además el más seguro. También es el más propenso a errores. En la mayoría de las ocasiones hemos optado por este método. Consiste en crear una estructura de datos no global (un registro) que recoge a todas las variables globales. Este registro, como es de esperar, tiene que ser pasado por argumento a todas y cada una de las funciones del código. La sobrecarga de este sistema no es excesiva pues sólo se precisan 32 bits extra del stack en cada llamada a una función.

De las dos opciones, quizás la más potente es la segunda por su independencia absoluta del runtime (y del formato del binario, ya que este método sólo sirve para formatos ELF) y por no necesitar ningún mecanismo de copia (en cuanto a las GOT se refiere). Tambien es cierto que el primer sistema es menos tedioso y propenso a errores.

3.4. Rendimiento del RUNTIME frente a MPI. El coste de la virtualización

La versión de la implementación de BICAV, basado en AMPI, que proponemos en este capítulo además de que mantiene el modelo de implementación típico de MPI, demuestra ser mucho más adecuado para la nueva generación de aplicaciones dinámicas y además no penaliza el rendimiento de aquellas aplicaciones que no precisan el dinamismo.

El concepto tras el que se apoya AMPI es la virtualización de procesos [181]. Un programa clásico MPI divide la computación en P procesadores y las implementaciones típicas basadas en MPI se limitan a ejecutar un proceso por procesador (salvo en los casos en que se decida hacer uso de MPI–2 donde en este caso podrá contemplarse la coexistencia de más de un flujo de control en el mismo procesador). En su lugar cuando se programa en el entorno AMPI la división de la computación se realiza sin tener que adaptarse al número de procesadores físicos existentes, se crean en su lugar tantos procesadores virtuales como se necesite. Procesadores que se repartirán entre los procesadores físicos existentes. En AMPI, *los procesos MPI se implementan como hebras de usuario, que embuten a su funcionalidad –en términos de objetos paralelos– y además disfrutan de la capacidad de poder ser migrados de un procesador físico a otro*, en un procesador físico pueden coexistir varios procesos de este tipo.

Las hebras empleadas por AMPI son hebras de usuario, es decir, que se crean y se planifican por código implementado por el usuario (no controlado por el sistema operativo), dadas las características de estas hebras es factible ejecutar miles de estas en un mismo procesador físico. El paso de mensajes entre estos procesos virtuales AMPI se implementa como comunicaciones entre estos objetos paralelos. Incluso tras una migración el runtime es capaz de redirigir los mensajes entre objetos de manera muy eficiente ([182], [183]).

Dadas estas características, en una comparación introductoria entre MPI y AMPI, podemos indicar que en el modelo de programación MPI, bloquear la CPU y quedar a la espera de que finalice una operación de E/S puede resultar en ejecuciones ineficientes. Especialmente en aquellas supercomputadoras donde el procesador de comunicaciones es suficientemente potente. Las llamadas no bloqueantes suelen emplearse en MPI para resolver esta situación. La solución consiste en permitir que podamos realizar otra computación que el proceso deba realizar y que no está directamente relacionada con el dato que se espera recibir. El problema suele residir en que no siempre tenemos *suficiente* computación dentro del mismo proceso. Además el precio a pagar por buscar rendimiento en este modelo de programación también debe considerar la

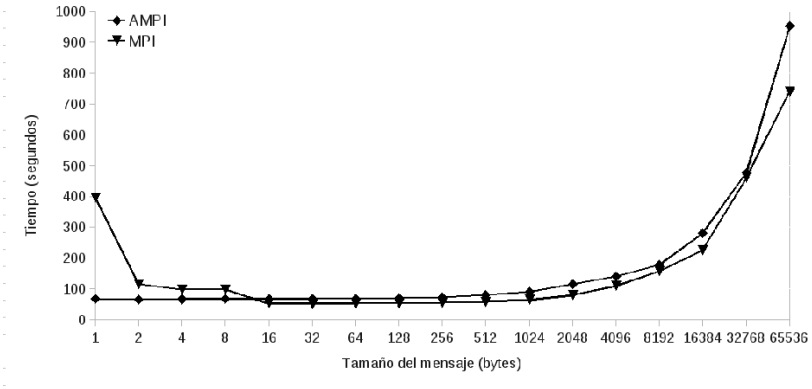


Figura 3.7: Test de comunicaciones 1: un sólo mensaje en el anillo.

complejidad añadida en la programación.

Para revelar la diferencia existente al usar uno u otro modelo, se han incluido datos de dos estudios realizados que comparan exclusivamente el paso de mensajes. Para los tests se ha empleado una arquitectura cluster de procesadores intel con conexión Gigabit Ethernet. Con estos test se pretende evaluar unicamente el coste del paso de mensajes. El escenario excluye necesariamente a las arquitecturas multicore, pues en estas la virtualización de AMPI la haría, como se verá más adelante, superior. Además se ha obviado toda computación. Los tests consisten en hacer circular mensajes entre los nodos. El primer test (figura 3.7) considera únicamente la circulación de un mensaje desde un procesador a otro con rank consecutivo, implementando una topología en anillo. En el segundo test (figura 3.8) se sigue respetando la topología en anillo pero en esta ocasión se envían tantos mensajes como procesadores hay en el anillo, también en orden de menor a mayor rank. En ambos tests la medición termina cuando el mensaje alcanza el procesador que lo inyectó. En ambos tests se ha repetido el envío del mensaje cien veces.

La figura 3.7 muestra cómo MPI se comporta mejor que AMPI para mensajes más grandes y en condiciones ideales –y existiendo únicamente un mensaje en circulación–, este resultado era de esperar pues en cada procesador se tiene a un proceso MPI o AMPI (que embute al proceso MPI equivalente). En este sentido y como AMPI constituye la capa superior del RTS, éste suele hacer uso del subsistema de comunicación disponible, añadiendo más intermediarios hasta poder transmitir que en el caso de MPI. Sin embargo en la figura 3.8 que se corresponde con el segundo test, AMPI experimenta una notable mejora frente a MPI. En la figura 3.8 se muestra cómo la existencia de varios mensajes en el anillo hace superior en rendimiento a la implementación orientada a objetos, el manejo interno de los mensajes es responsabilidad del RTS y este cuenta

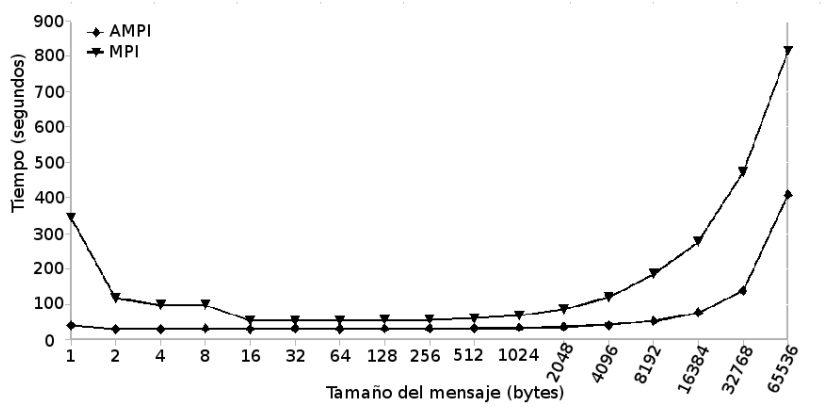


Figura 3.8: Test de comunicaciones 2: varios mensajes en el anillo.

con un planificador capaz de activar concurrentemente tareas dentro del mismo objeto, capacidad de la que carece la implementación de MPI. Para explicar las diferencias en los tests hay que considerar que el modelo de mensajes con el que trabaja AMPI emplea cabeceras en los mensajes, lo que supone un añadido que supera los 70 bytes. Además el runtime introduce latencias de 2 a 4 microsegundos en los mensajes de pequeño tamaño debido al cambio de contexto y a la actuación del planificador. La solución podría pasar por usar mensajes más grandes, pero en este caso se paga el precio de la característcia más apreciada que tienen las hebras en este entorno, esto es, la capacidad para migrar. Cada mensaje que se envía precisa de operaciones extra de copia para poder ser portado en caso de que la hebra migre. El peso del mensaje, por tanto, puede ser un inconveniente en lugar de una ventaja. No obstante, estos tiempos de sobrecarga pueden verse reducidos –como se ve en la figura 3.8– y mucho más si se trabaja con las librerías de optimización de comunicaciones que se desarrollan para este entorno de trabajo ([182] y [183]).

Como primera conclusión cabe indicar que el uso del entorno orientado a objetos no arroja rendimiento en condiciones ideales frente a las implementaciones de MPI. En condiciones normales (no ideales) los puntos de sobrecarga que padece el RTS pueden ser ocultados con una eficiente planificacion de subtareas. Los microsegundos que AMPI ha de pagar en las comunicaciones (que es el único punto donde la comparación entre AMPI y MPI se puede llevar a cabo) pueden ser obviados cuando la computación asociada al mensaje exceda varias centenas de microsegundos, lo que puede ser controlado a través del número de procesadores virtuales o VP (en definitiva hebras que contienen objetos que embuten procesos MPI). En los dos tests anteriores se ha tenido, siempre, un flujo de control por procesador (tanto en el caso MPI como

AMPI).

Para comprobar el rendimiento en el entorno de trabajo escogido, en relación a la granularidad del problema (número de VPs), implementamos la versión MPI de un algoritmo de relajación tridimensional (3D), donde el patrón de comunicaciones es muy similar al que sigue nuestra aplicación de reconstrucción de imágenes 3D (desarrollar programas similares a los que están en producción pero simplificados es una técnica extendida en paralelismo [133]) Este algoritmo, que llamamos Jacobi3D es un algoritmo iterativo que afecta a grupos de regiones en una malla. En cada iteración cada región de la malla intercambia parte de su información con sus seis vecinos inmediatos en la malla 3D. A su vez cada región realiza cálculos con los datos que recibe de sus vecinos. El algoritmo realiza una partición ***k*–cúbica** del conjunto de datos de modo que la implementación con MPI habrá de ejecutarse en una configuración determinada de procesadores mientras que la versión que emplea el framework AMPI no depende de ningún número de procesadores físicos. En la figura 3.9 podemos comprobar el comportamiento del algoritmo de relajación, para dimensiones del problema N=50 y N=100. Es necesario explicar que este algoritmo está preparado de tal forma que un aumento de VPs implíca un aumento igual del tamaño del problema, manteniendo constante la carga computacional de cada hebra (VP). En la respuesta se observa como desciende el rendimiento a partir de 64 procesadores virtuales, comprobando como este número de vp es el límite de granularidad.

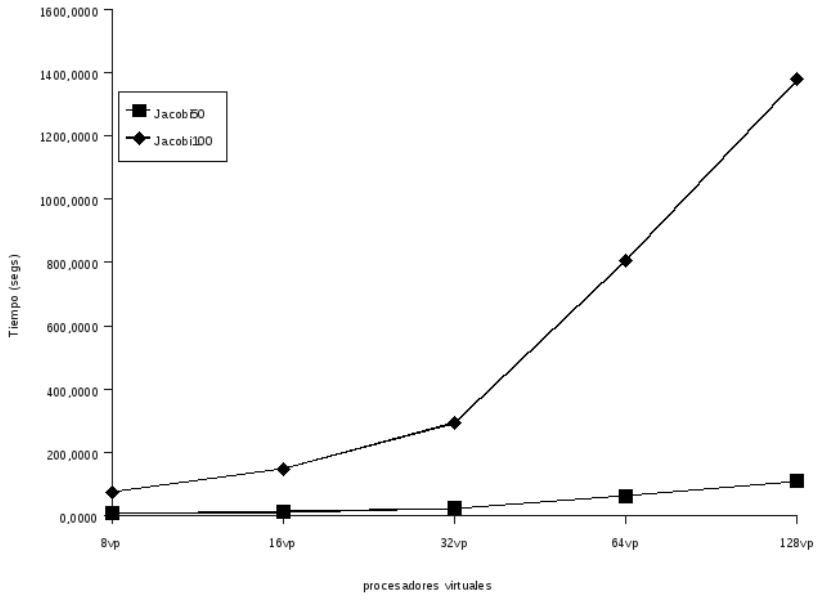


Figura 3.9: Virtualización: Algoritmo Iterativo.

La figura 3.9 muestra la aplicación sobre ocho procesadores físicos, pero virtualmente se está ejecutando en 8, 16, 32, 64 y 128 procesadores virtuales. Esto demuestra que la virtualización no hace al problema dependiente de los procesadores reales y por otro lado, que al doblar el problema no se duplica (como cabría esperar en MPI) el tiempo de ejecución. La siguiente configuración útil de procesadores reales para MPI sería 27 procesadores y el tiempo estimado de ejecución triplicaría en cada caso (Jacobi50 y Jacobi100) los valores de AMPI.

De acuerdo a las pruebas realizadas podemos concluir que el coste de la virtualización no crece con el número de procesadores virtuales, por tanto el coste que supone la virtualización se ve amortizado con creces por los beneficios que esta proporciona (gestión de la latencia en las comunicaciones). Es obvio que dependiendo del problema y de las condiciones del procesador, el número de procesadores virtuales tiene un límite. Como se observa en la figura 3.9 el límite es de 32 vp. Para representar gráficamente la información que queremos mostrar, se ha usado Projections [184] una herramienta JAVA de análisis de rendimiento *post mortem* de aplicaciones desarrolladas para el entorno AMPI.

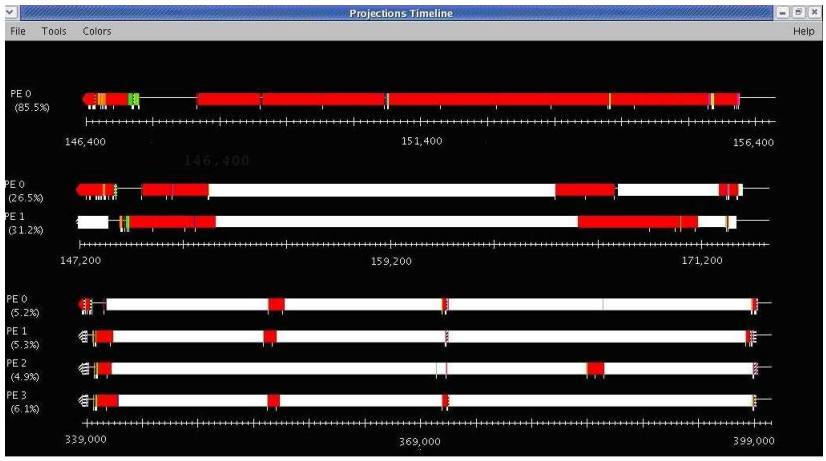


Figura 3.10: Ocultación de latencia y concurrencia

Como puede verse en la Fig.3.10, se ha experimentado usando tres escenarios distintos:

- Se ha reproducido el esquema de ejecución MPI, esto es un proceso por procesador, este es el caso en el que se usan cuatro procesadores (cada uno con una hebra). En este caso se puede ver (áreas en blanco) el tiempo desaprovechado en comunicaciones. La tabla 3.1 muestra el tiempo empleado en este caso (1vp/1p).
- Se han usado sólo dos procesadores físicos y se ha hecho *convivir* a dos procesadores

virtuales dentro de un mismo procesador físico. Se puede ver que el área dedicada a las comunicaciones se ve reducida. La aplicación ha sido avanzada durante los tiempos de comunicaciones. Ver tabla 3.1, caso (2vp/1p).

- Se ha usado sólo un procesador físico y en él se han ejecutado las cuatro hebras (procesadores virtuales que embuten a procesos MPI) y se puede ver que el procesador ha estado trabajado casi la totalidad del tiempo (salvo cambios de contexto) en nuestra aplicación. Ver tabla 3.1, caso (4vp/1p).

Tabla 3.1: Concurrencia. Efecto en Walltime (vp: procesador virtual, p: procesador)

1vp/1p	2vp/1p	4vp/1p
399ms	172ms	156ms

En una primera impresión se puede deducir que agrupar hebras en un mismo procesador genera un beneficio directo en el rendmiento, beneficio que se ve potenciado si estas hebras que se agrupan en un procesador físico comparten alguna propiedad de vecindad, impidiendo que las comunicaciones involucren a la red.

Cabe notar que el número de VPs por procesador no debe ser extremadamente alto, el rendimiento no crece linealmente con el número de procesos virtuales por procesador. Tener muchos procesos virtuales puede exceder la bonanza de rendimiento. Otro aspecto que favorece el rendimiento al usar procesos virtuales es la caché [185] ya que los procesos virtuales que residen en el mismo procesador fisico pueden aprovechar la localidad espacial lo que permitiría incrementar la eficiencia usando la comunicación entre hebras en el mismo procesador (intra-comunicación). Los beneficios de la caché son mucho más evidentes cuando el grano de la aplicación es más fino (vease el subapartado 3.6).

3.5. Estrategias de paralelización en reconstrucción 3D: Alternativa a MPI

Tradicionalmente las aplicaciones de reconstrucción han seguido un esquema *maestro-esclavo* e iterativo (se puede encontrar una caracterización de este tipo de aplicaciones en [133] –página 395–), quizás el único esquema aceptable en situaciones donde es imposible predecir los puntos de sincronización. En aplicaciones científicas, la irregularidad, ya sea del escenario de computación

o del problema en sí, hace poco aconsejable implementar un esquema síncrono. En este caso el esquema asíncrono sale elegido. La mejor forma de implementarlo es haciendo que la aplicación sea *iterativa* y que los puntos de sincronización se encuentren al final de cada iteración. Sería, un proceso maestro, quien tras cada punto de sincronización se encargase de restablecer la computación en su proceso iterativo.

BICAV es un algoritmo iterativo de reconstrucción basado en bloques que ha sido paralelizado siguiendo, por tanto, la aproximación SPMD (Single Program Multiple Data) [121]. En BICAV, el volumen a reconstruir se descompone en porciones (slabs) donde cada porción del volumen contiene un conjunto de *rebanadas o slices*. Estos *slabs* serán distribuidos entre todos los procesadores físicos (en el caso de la implementación MPI) o entre todos los procesos virtuales (en el caso de la implementación AMPI), vease la figura 3.11 (derecha), donde los slices son representados por las comlumnas. Los procesadores virtuales, tras esta descomposición, pueden trabajar cada uno en su subdominio, sin embargo, la interdependencia existente entre los slices que son vecinos (dado el uso de las funciones base esféricas -blobs) hace necesaria la inclusión de blobs redundantes en cada slab (columnas en gris claro en figura 3.11 (derecha)).

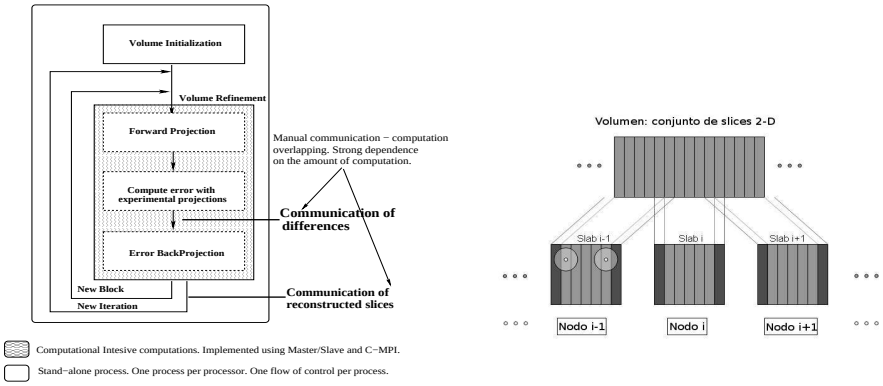


Figura 3.11: Algoritmo de reconstrucción iterativo (izquierda). Descomposición de datos para su paralelización (derecha)

Debido a esto se precisa que entre los procesadores exista intecambio de información (slices frontera) para permitir que slabs vecinos puedan actualizar los slices redundantes y así procesar correctamente la proyección y retro-proyección [121] (figura 3.11 (izquierda)). Estos puntos de comunicación suponen puntos en los que se hace necesario una sincronización. El ratio de comunicaciones depende del número de bloques usado y de las iteraciones que escojamos como limite en nuestra ejecución.

En cualquier proceso de paralelización donde existe comunicación entre los nodos, el tiempo empleado en las operaciones de E/S relacionadas con las operaciones de red se convierte en una traba que se debe evitar. La ocultación de la latencia entra en juego como único medio para resolver este efecto. La ocultación de la latencia consiste en solapar las operaciones de E/S (lugares donde se lleva a cabo la sincronización –ver figura 3.11 izquierda–) con computación. Es decir, en adelantar computación en los momentos en los que el procesador esté esperando a que se termine una operación de comunicación que involucra a la red. La propuesta de mejora en este aspecto viene dada por la implementación de la alternativa a la versión MPI de nuestro algoritmo de reconstrucción (ver figura 3.12). Esta propuesta se apoya en el modelo de hebras ofrecido por AMPI, hebras de usuario, característica que se emplea para ocultar el efecto de la latencia [143]. Las hebras de usuario que proporciona AMPI -cada hebra embute a un *proceso clásico MPI* ver figura 3.12- pueden conmutar de una a otra cuando la hebra en ejecución precisa detener el uso de la CPU para realizar operaciones de comunicación, es decir cuando la hebra alcanza uno de los puntos de sincronización de los que antes hemos hablado. Por tanto, mientras una hebra se

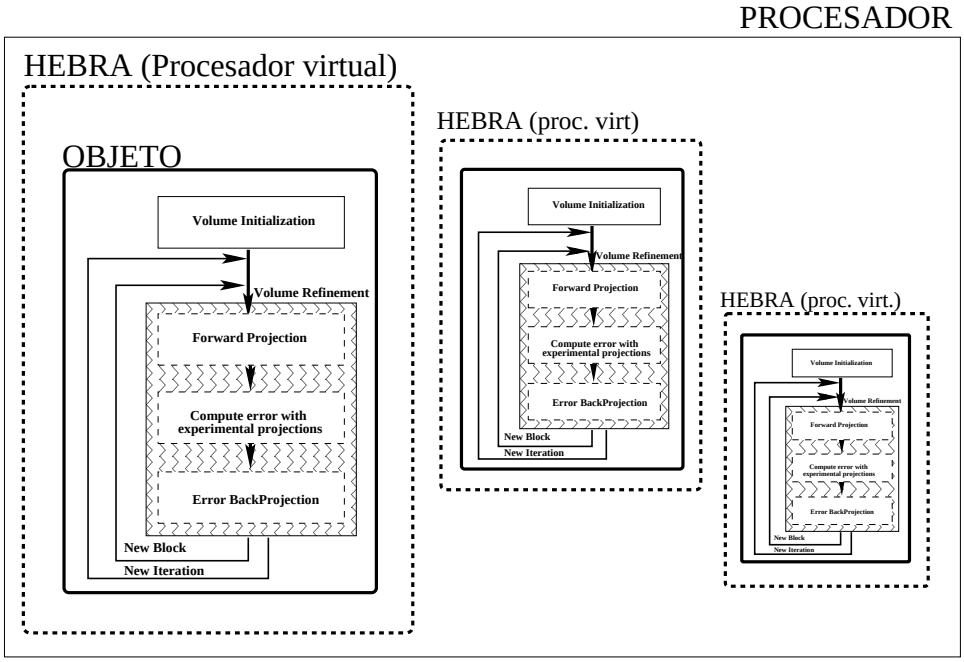


Figura 3.12: Propuesta de abstracción de procesos MPI usando el concepto de procesador virtual (hebra de usuario).

encuentra en espera, otra puede tomar el control de la CPU y evitar que esta gaste ciclos sin realizar operación alguna, avanzando así la aplicación. La mejora de rendimiento depende del algoritmo usado, de las características de la arquitectura de red subyacente y del soporte que el sistema ofrezca para operar el cambio de contexto. Como se ha mencionado anteriormente, la operación de reconstrucción mejora sus resultados al incrementar el número de bloques (de ecuaciones) usados. Por otro lado incrementar el número de bloques redunda en un incremento en las necesidades de comunicación, lo que constituye una causa de pérdida de rendimiento. La implementación AMPI propuesta sugiere que cada *slab* sea *embutido* dentro de cada hebra (ver figura 3.13), de tal modo que en cada procesador físico podramos tener tantos slabs como hebras instanciemos en él. Los patrones de comunicación entre nodos circundantes se mantendrá igual que en el caso de MPI ya que cada procesador físico habrá de comunicar el mismo número de slices redundantes tanto en el caso AMPI como MPI.

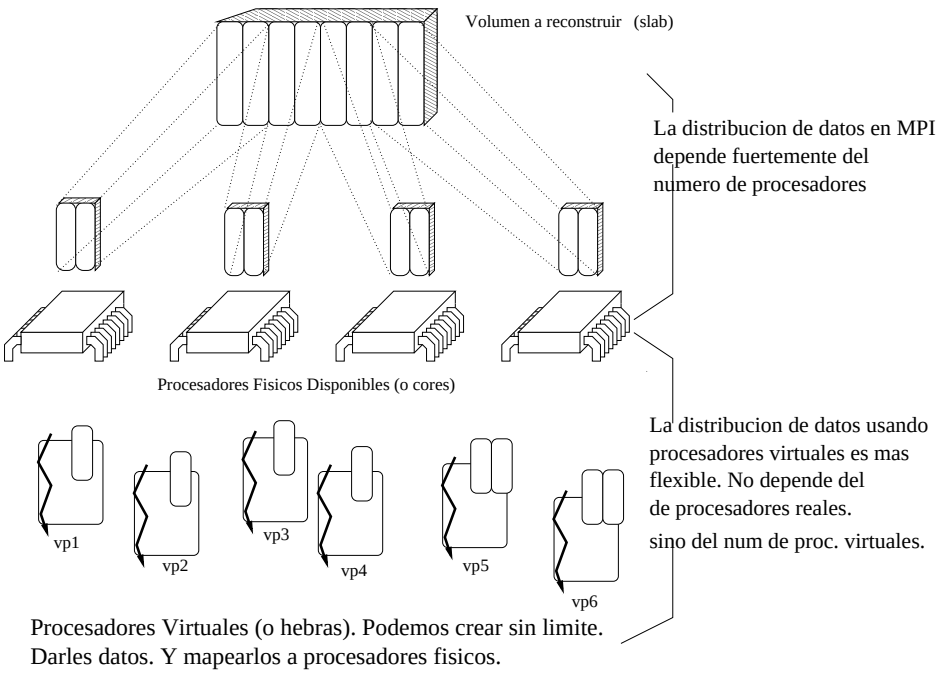


Figura 3.13: El volumen se descompone en grupos (slabs) de subvolúmenes (slices) que serán distribuidos entre los nodos

La gran ventaja de este modelo es que, dada la flexibilidad de las hebras de usuario, es posible *re–mapear* tantas veces como se desee el conjunto de hebras (o procesadores virtuales) en los procesadores físicos. De este modo se puede garantizar que la computación se adapte al entorno de ejecución.

3.5.1. Comportamiento de la aplicación en el entorno AMPI. Resultados

Se han empleado dos plataformas de computación diferentes para evaluar la presente implementación de nuestro algoritmo de reconstrucción. Una de las plataformas usadas ha sido uno de los clusters del NCSA, TUNGSTEN, que está formado por un servidor Dell poweredge 1750 que actúa como front-end con 3GB ECC DDR SDRAM y cuyos nodos de cómputo son Intel Xeon 3.2 GHz (32 bits), con 512 KB L2 Cache y 1MB L3 de Cache, todos los nodos estaban conectados con una red de 1 Gb Myrinet 2000. La segunda plataforma usada ha sido el cluster que nuestro grupo de investigación posee, VERMEER, que tiene como front-end a un Pentium dual IV XEON 3.2 Ghz, con 512KB Cache y monta una Gentoo Linux, la versión del Kernel es 2.4.32 SMP, cada nodo de cómputo posee dos Pentium IV XEON 3.06 Ghz, 512KB L2 Cache con 2GB of ECC DDR SDRAM, conectados por una red de 1Gb Ethernet.

Los tests están orientados a probar cómo se comporta la versión de la aplicación de reconstrucción que se ejecuta en el entorno multihebrado. Se pretende comparar la eficiencia de esta versión donde la ocultación de la latencia (comunicación / computación) se obtiene como consecuencia directa del entorno usado en contraste con la versión MPI donde esto se ha de implementar explícitamente. Se han lanzado tests de escalado en ambos sistemas, variando tanto el número de procesadores virtuales por procesador como el número de procesadores físicos, este último parámetro tanto para la versión MPI como AMPI.

Todos los resultados experimentales obtenidos provienen de dos grandes grupos de experimentos realizados con el fin de llevar a cabo la comparación entre métodos de implementación. En el primer grupo de experimentos se trabajó con el objetivo de medir la ganancia obtenida como consecuencia de la ocultación de latencia implicita que proporciona la implementación AMPI, en comparación con la implementación explícita utilizando MPI. Estos experimentos se han llevado a cabo en el cluster TUNGSTEN, el volumen a reconstruir ha sido de dimensiones 256x256x256 a partir de 70 imágenes de proyección de dimensión 256x256. El algoritmo empleado para la reconstrucción ha sido BICAV, se han usado 70 bloques (K=70) y 10 iteraciones. El número de procesadores se ha hecho variar de 1 a 64.

Explotar la concurrencia incluyendo más hebras o procesadores virtuales que representan cada uno la ejecución de un proceso de la aplicación Bicav convencional, se presenta en la tabla 3.4 donde se recogen tiempos de ejecución de la aplicación implementada usando AMPI, contrastando los resultados con los obtenidos mediante la implementación MPI (tablas 3.2 y 3.3)

# processors	CPU	Walltime	Ratio	SpeedUp
2	13291,50	13330,63	0,9971	1,99
4	6747,83	6823,23	0,9889	3,90
8	3400,28	3515,21	0,9673	7,56
16	1730,66	1933,78	0,8950	13,75
32	871,35	2007,40	0,4341	13,24
64	466,10	2510,05	0,1857	10,59

Tabla 3.2: Bicav MPI sin ocultación de latencia (v256, iter 10, K 70)

#processors	CPU	Walltime	Ratio	SpeedUp
2	12847,58	12861,93	0,9989	2,00
4	6529,53	6555,73	0,9960	3,92
8	3305,14	3342,46	0,9888	7,69
16	1685,37	1721,89	0,9788	14,92
32	935,16	1797,20	0,5203	14,30
64	471,65	2309,95	0,2042	11,12

Tabla 3.3: Bicav MPI con ocultación de latencia (v256, iter 10, K 70)

en la reconstrucción de un volumen de 256x256x256, usando un valor de bloques típico (K=70) y 10 iteraciones. En el caso de MPI se recogen dos versiones, la versión que no implementa ocultación de la latencia y que por lo tanto está expuesta a la pérdida de rendimiento y la versión que implementa la ocultación de la latencia. En la tabla 3.2 se presenta la aplicación Bicav sin ocultación de latencia incluída y en su versión MPI. En esta tabla se puede apreciar que cuando se ejecuta la aplicación sobre dos procesadores, la diferencia entre el tiempo que la CPU está realizando operaciones frente al tiempo que está realizando comunicaciones (ratio) es mínima y esto es bueno ya que la CPU se ha estado encargando en un 99,7 % de resolver el problema de la reconstrucción. En la tabla 3.3 donde se puede observar que con dos procesadores se emplea prácticamente todo el tiempo 99.9 % de ejecución del programa en resolver el problema y sólo un 0.01 % se emplea en comunicaciones que no han podido ser *ocultadas* con computación. En este caso donde en la versión con ocultación de latencia y la versión que no tiene ocultación de latencia se ejecutan en dos procesadores, la diferencia es mínima y quizá no se justifique el esfuerzo de realizar la implementación que se encargue de ocultar latencias. Si se estudia de nuevo la tabla 3.2 se puede observar como si se intenta escalar la aplicación a más procesadores (y por tanto se incrementa el porcentaje de uso de la red) el ratio entre el uso de CPU y las comunicaciones (respecto del tiempo de ejecución total) informa datos cada vez más negativos.

Si la aplicación Bicav se lleva a 16 procesadores, como se puede observar en la tabla 3.2 las comunicaciones únicamente se solapan en un 89,5 % quedando el resto del tiempo sin poder ser usado para realizar otros cálculos y por tanto afectando de manera directa al tiempo de ejecución total. Si la aplicación se porta a 32 procesadores, la caída de rendimiento es drástica y se justifica plenamente la implementación de las técnicas de ocultación de la latencia.

En la tabla 3.3 se presenta un análisis de la evolución, según se añaden nodos a la computación, en este caso se puede observar como se consigue un aprovechamiento prácticamente ideal de los tiempos de comunicación (para realizar cómputo) ya que en el caso de dos procesadores se llega incluso a tener un 99,9 % de solape entre comunicaciones y computación. No obstante, al incrementar el número de procesadores se va experimentando una pérdida (aunque menos notable que en el caso de no incluir latencia) de rendimiento. Tal y como muestra la tabla 3.3, al pasar a 32 procesadores las comunicaciones entre ellos son tal que no se encuentra computación suficiente (ver sección 6.2 del capítulo 2) para poder amortizar las comunicaciones. Una solución posible a este caso es hacer que el problema, al escalar, crezca en computación (procese más datos) cosa que en Bicav no es posible, aunque si es posible hacer que con la versión AMPI, al escalar podamos tener mayor número de procesadores virtuales por procesador físico y de este modo amortizar las comunicaciones.

Disponer de varios procesadores virtuales puede favorecer la ocultación de la latencia y el permitir así la escalabilidad del problema, las siguientes tablas estudian cómo afecta el incremento de estos procesadores virtuales al problema de la reconstrucción. En la tabla 3.4 se puede comprobar como con dos procesadores virtuales se alcanza el mejor speedup, no mejorando éste al aunmentar el número de VPs. Una explicación que justifica este comportamiento está en que en el sistema utilizado (TUNGSTEN) el runtime usado como la red de comunicación están optimizados. En el caso del runtime estamos usando opciones de compilación que optimizan las comunicaciones para la arquitectura de red Myrinet, haciendo que nos quedemos pronto sin *slots* de comunicaciones donde poder acomodar la computación de los procesadores virtuales.

La figura 3.14 muestra gráficamente que el uso de dos procesadores virtuales es en este caso la condición más favorable que permite explotar eficientemente la concurrencia y que por tanto la ocultación de latencia implícita supone una mejora tangible en el speedup.

En la figura 3.15 se representa la evolución del SpeedUp para las tres implementaciones analizadas, MPI con y sin ocultación de latencia y AMPI. En la figura 3.16 se puede observar

# Procesadores	Virtual Processors	CPU max	Walltime	Ratio	SpeedUp
	2	12469,63	12478,16	0,999	1,999
2 p	4	12441,52	12444,66	1,000	**2,004**
	8	12449,82	12464,04	0,999	1,987
	4	6420,73	6440,77	0,997	3,872
4 p	8	6337,55	6348,91	0,998	**3,928**
	16	6354,82	6359,91	0,999	3,875
	8	3223,6	3249,85	0,992	7,674
8 p	16	3204,12	3212,48	0,997	**7,763**
	32	3246,47	3258,46	0,996	7,489
	16	1685,28	1708,67	0,986	14,596
16 p	32	1660,76	1666,05	0,997	**14,969**
	64	1690,38	1707,03	0,990	14,610
	32	926,07	939,22	0,986	26,55
32 p	64	909,56	914,13	0,995	**30,38**
	128	1011,97	1019,11	0,993	24,472
	64	494,44	517,73	0,955	48,17
64 p	128	402,80	410,25	0,982	**60,79**
	256	503,55	521,19	0,966	47,85

Tabla 3.4: Bicav AMPI con ocultación de latencia (v256, iter 10, K 70). Incrementando el número de procesadores virtuales

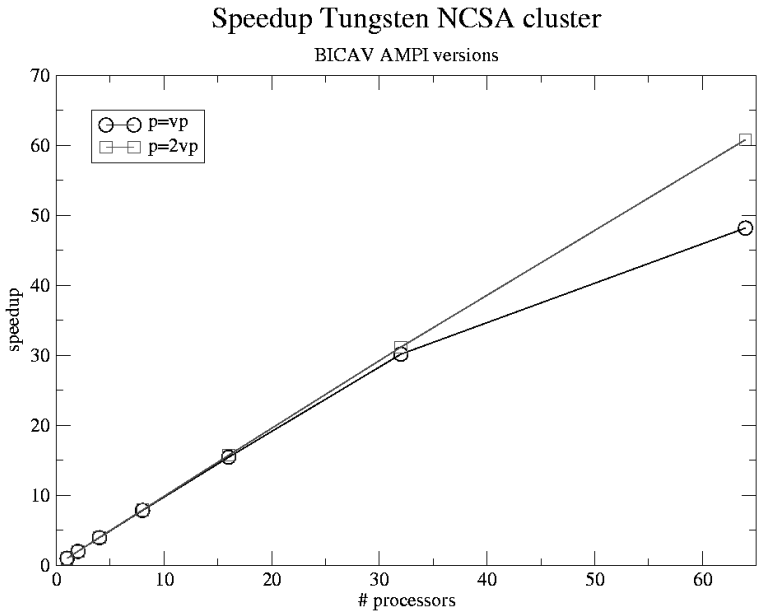


Figura 3.14: BICAV. SpeedUps en el cluster Tungsten.

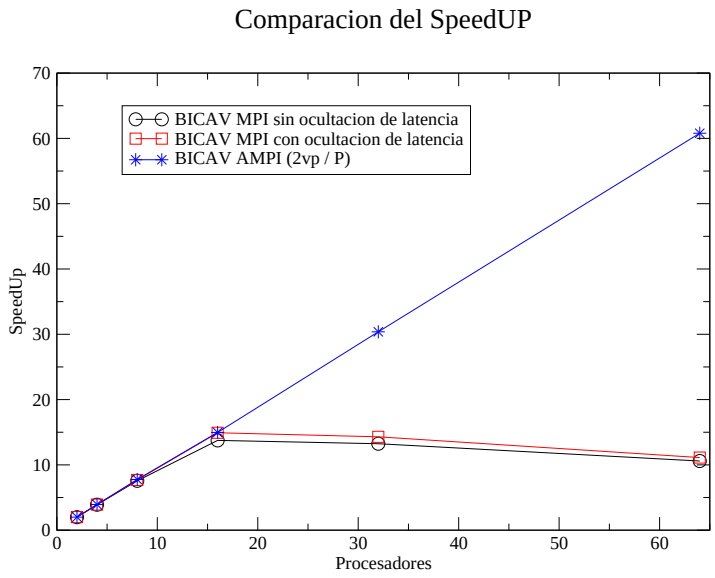


Figura 3.15: BICAV. Evolución en el cluster Tungsten.

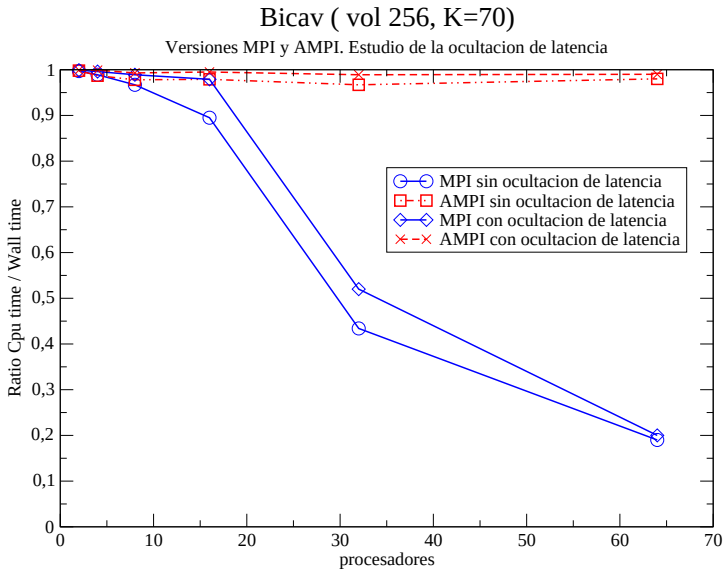


Figura 3.16: Efecto de la virtualización AMPI en la ocultación de latencia. Comparación con las versiones MPI de Bicav.

el ratio obtenido por las versiones MPI y AMPI. En este caso, AMPI muestra una ocultación en términos generales mejor que la obtenida con la versión MPI con ocultación de latencia (tabla 3.3). La tabla 3.4 muestra cómo incluso en el caso de 32 procesadores se consigue una ocultación de latencia que ronda en todos los casos el 99 %. Esto se debe a que siempre que ocurre una comunicación (un procesador virtual inicia la comunicación de sus slices edge) podemos conmutar a otro procesador virtual.

La segunda tanda de experimentos fueron realizados con la intención de cuantificar la influencia del número de bloques usados en el algoritmo de reconstrucción respecto del rendimiento global de la aplicación. Y sobre su consecuencia directa sobre el escalado de la aplicación. Al incrementar el número de bloques, la convergencia del algoritmo se acelera. Incrementar el número de bloques también supone un importante incremento en las comunicaciones. Un incremento de las comunicaciones hace necesario, en aras del rendimiento, dos cosas:

- Computación suficiente para rellenar los espacios destinados a la comunicación.
- Un medio para poder encontrar computación hábil para el fin anterior.

En este escenario, la versión MPI posee la implementación de la ocultación de latencia empleando llamadas asíncronas para las comunicaciones y aprovechando fases de comunicación para anticipar computación (figura 3.11). Incrementar las comunicaciones hace que la versión MPI acuse el efecto adverso de las mismas antes que la versión AMPI, versión que mantiene las cotas de rendimiento esperadas. Las ejecuciones que han dado lugar a los resultados que aquí se presentan se han realizado sobre el cluster VERMEER, empleando hasta un máximo de 32 procesadores físicos. Se han usado dos *datasets o volúmenes* diferentes que suponen dos tamaños de problema diferente para la reconstrucción. Primero se abordó la reconstrucción con un volumen de 256x256x256 a partir de 256 imágenes de proyección de dimensiones 256x256 (figura 3.17).

En segundo lugar se empleó un dataset para la reconstrucción de un volumen de 512x512x512 a partir de 512 imágenes de proyección de 512x512 (figura 3.18). El algoritmo BICAV, tanto en la versión MPI como en la versión multihebrada AMPI ha sido probado en ambos casos con sendos datasets. Respecto al número de bloques K, en este escenario de ejecuciones, se han seleccionado dos valores por ejecución. Uno de ellos con el número mayor de bloques posibles en cada caso, con el fin de acelerar la convergencia y el otro valor de K que se usó fué un valor intermedio. Tanto la

# Procesadores	Virtual Processors	CPU max	Walltime	Ratio
2 p	2	1018,1	1022,97	0,9952
	4	1022,17	1023,83	0,9984
	8	1025,87	1033,92	0,9922
	16	1018,12	1018,97	0,9992
	32	1013,8	1014,78	0,9990
	64	1014,42	1014,4	1,0000
	128	1012,36	1013,2	**0,9992**
4 p	4	512,09	523,73	0,9778
	8	512,65	513,1	0,9991
	16	512,79	512,94	0,9997
	32	509,61	510,22	0,9988
	64	509,23	509,6	0,9993
	128	509,24	509,27	**0,9999**
8 p	8	254,02	262,83	0,9665
	16	253,99	254,21	0,9991
	32	255,64	256,54	0,9965
	64	256,39	257,02	0,9975
	128	256,51	256,55	**0,9998**
16 p	16	128,21	144,75	0,8857
	32	128,63	130,13	0,9885
	64	129,91	131,57	0,9874
	128	130,13	131,1	**0,9926**
32 p	32	65,97	67,77	0,9734
	64	66,25	67,64	0,9795
	128	66,99	67,2	**0,9969**

Tabla 3.5: Efecto del número de VPs en el ratio.

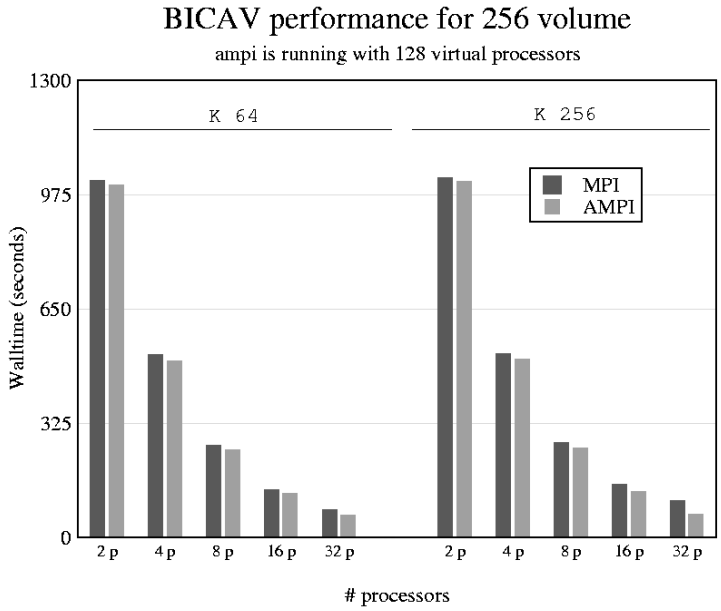


Figura 3.17: Tiempos de ejecución por iteración en BICAV.Volumen 256x256x256

figura 3.17 como la figura 3.18, muestran el valor del *walltime* de la versión MPI frente a la versión AMPI, esta última empleando 128 VPs, por ser esta, aunque ligeramente, la configuración que mejores resultados da en todos los casos, según podemos observar en la tabla 3.5. En esta tabla se representa un análisis de la evolución del rendimiento para el primer conjunto de datasets al variar el número de procesadores virtuales (VPs). Es interesante observar que por debajo de *K=64* ambas versiones (MPI y AMPI) parecen comportarse de manera practicamente idéntica, mostrando una cierta ventaja en favor de la versión AMPI. Usar un número de bloques superior a 64 supone una mejora notable para el caso de la versión AMPI especialmente en ejecuciones que usen más de 16 procesadores. En este caso se dejan ver el beneficio de explotar la concurrencia oculta a través del solape adaptativo. En el caso de la reconstrucción del volumen de dimensión 256x256x256 AMPI se comporta mejor cuantos más procesadores se vayan incluyendo en la reconstrucción. Esto es, cuanto mayor sea el uso de la red.

En el caso que expone la figura 3.18 se experimenta la misma tendencia vista en la figura 3.17. En este caso la computación es mayor que en caso anterior, por lo que siempre hay computación disponible para ocultar las latencias, de modo que la versión AMPI se comportará mejor que la MPI incluso cuando se añadan más procesadores. En el caso anterior, las latencias de red pueden llegar a superar a los slots de computación.

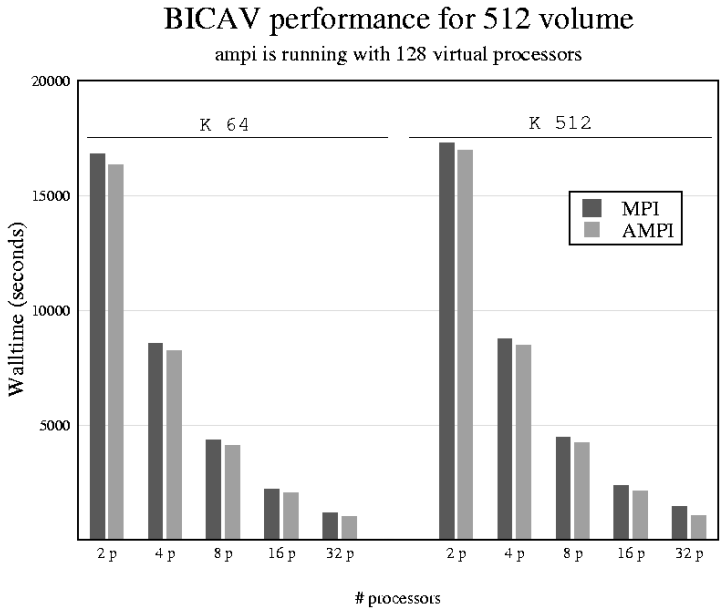


Figura 3.18: Tiempos de ejecución por iteración en BICAV. Volumen 512x512x512

En las figuras 3.19 y 3.20, se muestra la ganacia (speedUp) obtenida por la versión multihebrada de BICAV usando 128 procesos virtuales, para los dos valores de K usados y para los dos datasets. Las curvas de speedup muestran claramente que la aplicación implementada con MPI pierde la linealidad en el speedup cuando se emplean más de ocho procesadores. Sin embargo, AMPI se mantiene en un speedup practicamente lineal incluso al ejecutar sobre los treinta y dos procesadores físicos. El hebrado y su rápido cambio de contexto están, sin duda, favoreciendo esta consecución del SpeedUp casi óptimo.

En la tabla 3.6 y en la tabla 3.7 se presenta la diferencia relativa (columnas %) existente entre el *cputime* y *walltime* para ambos tamaños del problema. Estas tablas muestran los casos en los que tanto para el volumen de dimesión 256 como el volumen de dimensión 512 se tratan con el máximo valor de K (K=256 y K=512, respectivamente). Es decir, los casos donde se alcanza la máxima tasa de comunicación posible para ambos volúmenes. Es fácil comprobar cómo para AMPI los tiempos walltime y cputime son prácticamente idénticos (los tiempos de cambio de contexto y gestión de la hebra hacen que los dos tiempos no sean exactos). Que estos tiempos sean tan similares demuestra que la CPU ha estado prácticamente el 100 % del tiempo en uso, incluso durante las etapas de comunicación. Estos resultados contrastan con los obtenidos por la versión que usa la implementación MPI, caso en el que las divergencias entre ambos tiempos

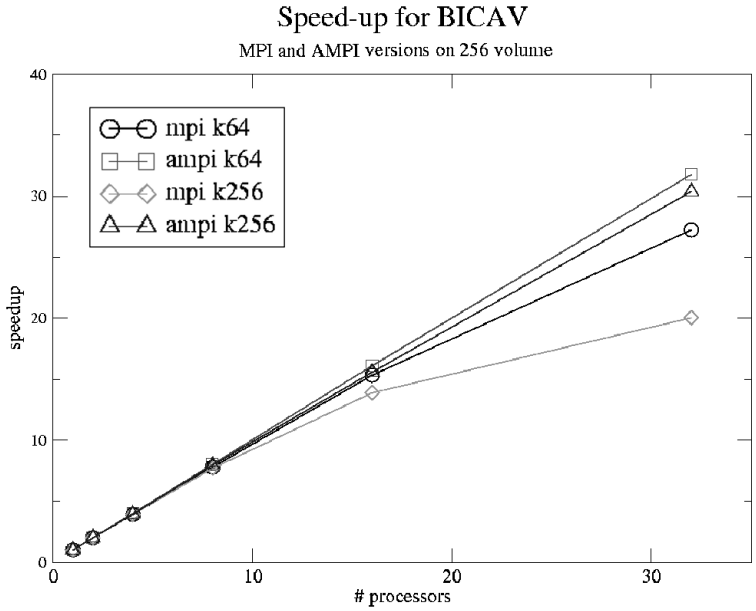


Figura 3.19: SpeedUp en el cluster VERMEER para el volumen de dimensión 256

Tabla 3.6: Diferencias entre el tiempo de CPU y WallTime. Concurrencia (k256)

K 256 (volume 256)						
	MPI			AMPI		
Procs	CPU	WALL	%	CPU	WALL	%
2	991	1021	2.9	1012	1013	0.1
4	502	520	3.5	509	509	0
8	251	265	5.6	257	257	0
16	126	147	17.1	130	131	0.8
32	63	192	62.4	67	68	1.5

(cputime y walltime) se acusan más. A la interpretación de los resultados expresados en esta tabla hay que añadirles el hecho de que durante la ejecución de los experimentos no hubo otras operaciones de E/S ni llamadas a métodos/funciones que no fueran del propio algoritmo, lo que nos permite afirmar que la versión multihebrada para la implementación del algoritmo BICAV se aprovecha la concurrencia (posible gracias a los huecos de comunicación) al máximo.

Como aspecto destacable, cabe citar que al incrementar el número de bloques en la aplicación se obtiene mejor reconstrucción pues el volumen reconstruido ha sido refinado más veces (ver capítulo 2). Esto conlleva que las comunicaciones crecen en comparación con la computación. También es cierto que crecen la computación. Por lo que la implementación de técnicas de ocultación de latencia manuales puede llegar a limitar las ganancias de rendimiento en recons-

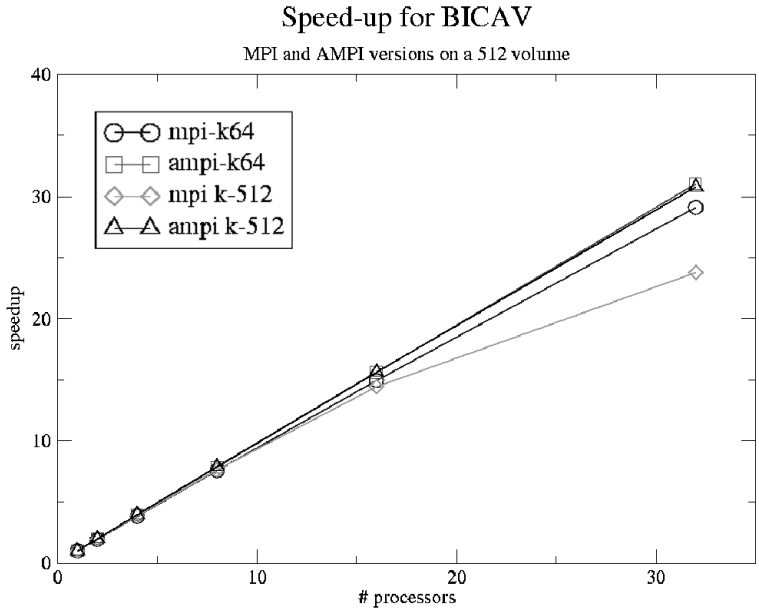


Figura 3.20: SpeedUp en el cluster VERMEER para el volumen de dimensión 512

Tabla 3.7: Diferencias entre el tiempo de CPU y WallTime. Concurrencia (k512)

K 512 (volume 512)						
	MPI			AMPI		
Procs	CPU	WALL	%	CPU	WALL	%
2	16749	17217	2.8	16989	16988	0.0
4	8432	8705	3.2	8479	8481	0.0
8	4222	4418	4.7	4251	4263	0.3
16	2128	2334	9.7	2138	2154	0.7
32	1071	1417	32.3	1092	1094	0.2

trucciones donde se precise alta resolución. En cierto modo, se puede pensar que al incrementar el número de procesadores se puede incrementar el número de bloques (siguiendo el modelo de la ley de Gustafson [60]) de modo que podamos incluir más procesadores virtuales, de modo que la concurrencia de estos ayude a explotar mejor la ocultación de latencia y como beneficio directo, la reducción del efecto de las zonas de código secuenciales existentes en BICAV.

En cuanto a un estudio del uso que ambas versiones hacen de la memoria virtual del sistema, se han realizado medidas (usando *vmstat*) de la cantidad de memoria libre registrada –en media– en cada procesador de los que componen el mapa de ocho procesadores usados. Cabe citar que el uso de la versión virtualizada de BICAV (versión hebrada) gestiona la memoria mejor gracias al RTS en contraste a la gestión que experimenta la versión basada en procesos (también hay que citar que la encapsulación, propiedad característica de los objetos, en los que se traduce un proceso aprovechan mejor la jerarquía de memoria[185]).

En la versión basada en MPI, es el propio proceso quien mediante llamadas al sistema operativo debe conseguir la memoria dinámica necesaria para sus tareas. En el caso de la versión multihebrada AMPI, el RTS hace la reserva de memoria y se la proporciona a cada proceso virtual cuando se la pide. De esta manera si en un procesador hay más de un proceso virtual (VP1 y VP2) , cuando el proceso VP1 requiera memoria realizará una llamada al RTS y se activará la ejecución de VP2 mientras VP1 es servido (esto se puede apreciar en la reserva escalonada que muestra la figura 3.21 cuando son 8 procesos virtuales los que tenemos por procesador físico).

En la citada figura 3.21, se aprecia que en aquellos nodos donde se lanza la aplicación BICAV implementada en MPI y C se puede aprecia más memoria libre (media) por nodo –disponible para el uso del proceso BICAV– que en los casos en los que se lanza BICAV hebrado. El motivo es que en el caso del proceso MPI, éste alberga en su mapa de memoria las librerías de mpi, libc, las funciones de reconstrucción, etc.... En el procesador con BICAV hebrado, cada proceso virtual tiene lo mismo, pero en cada procesador hay además un proceso extra que les da soporte (a todos los procesos virtuales, el RTS) y esto se nota en el consumo de memoria. La ventaja es que la naturaleza reentrante del RTS permite que los procesos virtuales no deban incorporar funcionalidad extra, siendo mucho más ligeros que un proceso MPI. Si se observa la figura 3.21, se podrá ver que en los nodos con MPI queda libres más memoria que con la versión hebrada. Esto es motivo del RTS, pues cuando se lanza la versión en la que únicamente hay 1 proceso virtual por procesador, la única diferencia con la versión MPI es el RTS que gestiona ese proceso virtual.

¿Merece la pena doblar el consumo de memoria? En este caso, sí. Pues en una comparación uno a uno, se observa que el proceso virtual termina antes que el proceso MPI. Y en el caso de iniciar 8 procesos virtuales por procesador, aún merece más la pena, pues el impacto de tener ocho flujos de control (en memoria, comparado con únicamente tener un flujo de control virtual) apenas es relevante y la ganancia en tiempo de ejecución es notable.

¿Es comparable la memoria que consumirían ocho procesos MPI reales en un procesador con la memoria consumida por los ocho procesos virtuales en un procesador? Pues quizás no, pero la ejecución concurrente de los procesos sería torpe y degradaría el rendimiento. Trabajos como el presentado recientemente en [133] demuestran que el mejor mediador entre procesos MPI ejecutados en el mismo procesador es un RTS. En este caso se justificaría el consumo que experimentamos, tanto por la celeridad en la ejecución como por la gestión que se realiza de la memoria.

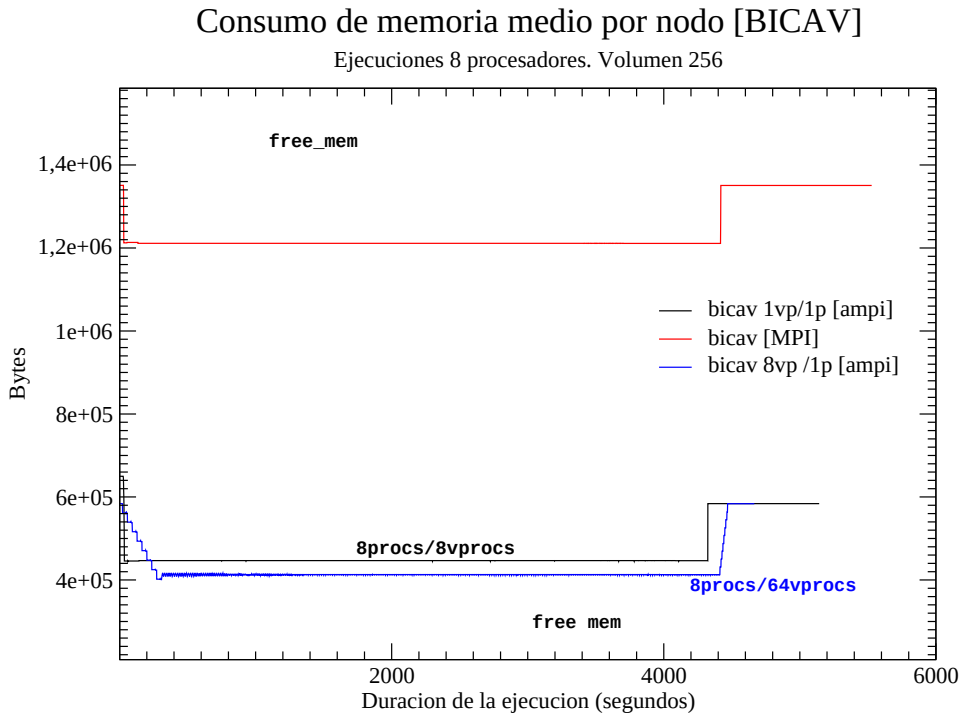


Figura 3.21: Memoria libre: versión MPI vs versión AMPI

3.6. Implementación de la reconstrucción usando un modelo orientado a objetos de grano más fino

Los modelos de programación en uso para trabajar con procesadores multicore y arquitecturas basadas en GPGPUs apenas consideran los avances experimentados en las plataformas transputers [157], [162] y [164] (incluso cuando la semejanza entre transputers y multicores / GPGPUs es muy grande).

La arquitectura de la plataforma en la que se despliega la aplicación afecta directamente sobre el rendimiento de la misma. En el caso de los multicores la situación es mucho más delicada pues son arquitecturas semejantes pero donde los detalles arquitectónicos como los relacionados con las cachés, jerarquías de memoria, y similares no siguen un estandar, de modo que podemos encontrar diferentes experiencias, en cuanto al análisis de rendimiento. Por ejemplo, los dos núcleos de una arquitectura UltraSparc IV son prácticamente independientes (el uno del otro) salvo por ciertas zonas comunes que se encuentran fuera del chip, por otro lado, el procesador PowerP4 tiene dos núcleos con la caché L2 compartida para facilitar una ágil comunicación entre hebras que residen en núcleos diferentes. De esto, se puede afirmar que, al menos en lo que concierne a los multicore, disponer de una capa de abstracción sería bastante útil por lo menos a la hora de disponer de un medio a través del que se puedan realizar portaciones transparentes. Aún con estas capas de abstracción hay que tener en cuenta que la reducción de la cantidad de memoria disponible dentro del chip, para cada hebra, el acceso compartido por los núcleos a memoria principal así como la necesidad de conservar memoria y acceder a la información que en ella reside de modo que respete la jerarquía existente son las piedras angulares de todos los desarrolladores de aplicaciones para estas plataformas.

Dadas las patentes dificultades y coste de la paralelización manual usando un modelo de programación de bajo nivel [174], [140], se hace necesario disponer de modelos de programación y entornos que ofrezcan, de manera razonable, la capacidad de adaptar código ya existente así como ofrecer la posibilidad de crear nuevas aplicaciones paralelas. La paralelización automática (como se indicó antes) es una característica deseable pero extremadamente dificil de conseguir, de hecho es motivo de numerosos trabajos de investigación que avanzan con interesantes progresos, sin embargo, la orientación a objetos ofrece una sencilla y directa forma de expresar interacciones (de expresar paralelismo). Un programa orientado a objetos es, en cierto modo, *un programa paralelo.*

Los modelos de programación paralela diseñados para la computación de altas prestaciones (o HPC) como es el caso de MPI, deberían ser implementados (cuidadosamente) para arquitecturas multicore, no obstante, no parece que sean un modelo apropiado para la portación de aplicaciones [140] pues se precisa (tanto en MPI como en modelos similares) una reorganización del código, en el caso particular de MPI se precisa la reescritura del programa para obtener el *código local* y por tanto no facilitan la paralelización incremental. Suelen ser prolijos en el consumo de memoria. Los inconvenientes de las jerarquias de memoria pueden incluso dificultar que se explote el rendimiento de las cachés compartidas.

En el caso de las arquitecturas multicore (aunque no hay una clara estandarización entre diferentes procesadores multicore) existen un par de características a subrayar :

- La inclusión de varios cores en el chip no ofrece ganancias directas a aplicaciones convencionales.
- Todos los núcleos usan el mismo espacio de memoria compartida, lo que invita a pensar en el modelo de programación multihebrada.

La primera característica supone un cambio en el modelo de programación, por diferentes razones :

- La escala de integración ha hecho posible que se puedan integrar dos núcleos en un mismo chip, de modo que la capacidad computacional puede crecer en la medida en que sea posible integrar más núcleos por chip (a mayor número de núcleos, más capacidad de computación puede ofrecer el chip). La frecuencia por núcleo, como consecuencia, se verá reducida.
- Cachés compartidas y memoria. El modelo de programación que se use deberá hacer un uso adecuado de las cachés para evitar caídas en el rendimiento debido a conflictos. Además se deben implantar técnicas de contención en el acceso a memoria (accesos transaccionales, atomicidad, . . .).
- Habilidad para mantener más de un flujo de control en ejecución. Tener varios núcleos implica que el programador puede (y debe) mantener varias ejecuciones activas. El beneficio para la computación de altas prestaciones (HPC) es evidente, ya que es posible realizar más de una tarea en paralelo, con lo que la labor del programador se complica más. El uso de las abstracciones es recomendable.

Los modelos de programación multihebrada son útiles para mantener varios flujos de control activos [154], logrando –en cierta medida– salvar la reducción en las frecuencias de reloj, no obstante esto no soluciona el hacer un uso eficiente de las cachés, de los accesos a memoria y tampoco plantea una solución a la adaptación de las aplicaciones de HPC ya existentes a las nuevas arquitecturas. Es en este punto donde se precisa hacer uso de abstracciones para poder mantener las ganancias de las aplicaciones HPC [151]. El paradigma de programación Orientada a Objetos ofrece un medio flexible para describir cómo llevar a cabo la computación. El modelo de objetos usado debe poder conjugar concurrencia y paralelismo. El paradigma Orientada a Objetos es inherentemente paralelo [157]. Diseñar una aplicación Orientada a Objetos supone poder dividir a la computación en entidades autónomas (objetos) capaces de intercambiar información. Incluir la concurrencia en este modelo supone que una de estas entidades autónomas pueda hacer cuasi–simultáneamente varias tareas, en este caso y para tal fin se recurre a un modelo de objetos *Parallel Objects* [165].

Desde un principio, los sistemas orientados a objetos dejaron de lado las características relacionadas con el paralelismo (el paradigma Orientado a Objetos no ha explotado en demasía esta capacidad). Sin embargo, no ocurrió lo mismo con todo lo relacionado con la concurrencia. En este caso, la orientación a objetos según [186] ha ganado bastante terreno; la Orientación a Objetos facilita la descomposición del problema en componentes segregados (dominios de ejecución disjuntos) lo que hace posible tener a entidades de computación independientes ejecutándose en un único procesador (aprovechamiento de las cachés, acceso transaccional a la memoria, atomicidad, etc), son conceptos muy desarrollados en la Orientación a Objetos a través de aspectos propios como el *encapsulamiento, etc*. Los sistemas basados en la orientación a objetos definen dos tipos de interacciones:

- Inter-Objeto: deriva de la consideración de que los objetos son entidades computacionales autónomas. La computación se traduce en una *agregación* de objetos organizados en una *composición* (aspecto en el que hemos basado nuestas implementaciones). Los objetos se comunican a través de la invocación de métodos, de igual modo que si fueran proveedores-consumidores de un servicio. Esta interacción conlleva un indeterminismo global al que contribuye el tiempo en servir las peticiones.

- Intra-objeto: guarda relación con la concurrencia dentro del objeto mismo (como ocurre con el modelo de actores). Los sistemas de objetos concurrentes proporcionan paralelismo

intra-objeto. El indeterminismo local, propio de este tipo de interacción aparece ante la tarea de decidir la siguiente tarea a ejecutar. El modelo de objetos propuesto en [165] evita esta incoherencia a través de la herencia múltiple y de las características dinámicas propias del modelo de objetos. La orientación a objetos que se emplea en todo este capítulo se apoya en un modelo de objetos [144] construido sobre el patrón *actor*[187].

Algunas de las características del modelo de objetos que son de interés para estas arquitecturas (y que personalmente creo también de interés para las plataformas cluster, multicore y las basadas en GPUs) se listan a continuación:

- Los objetos protegen sus datos internos.
- Los objetos protegen su estado interno y no lo comparten. La interfaz que un objeto hace pública o visible se limita en exclusiva a las operaciones públicas que manejan su estado interno. Los objetos son por tanto abstracciones de datos[9].
- Un objeto es una instancia de un tipo abstracto de datos (su clase). Además su comportamiento (conjunto de servicios o métodos) también se deriva de su clase. Cualquier clase contiene la descripción de cada instancia. Además las relaciones de herencia facultan la factorización.

Las dos primeras propieadades definen un escenario basado en entornos separados y locales. Desde esta perspectiva, los objetos pueden ser ejecutados en paralelo, al igual que los actores [187]. Por tanto, los objetos son unidades estructurales y unidades concurrentes. Cada objeto es el propietario de una cola privada de mensajes en la que se almacena cada petición de servicio (invocación de método) que se le dirije. Cada objeto ejecuta una tarea cada vez.

Los sistemas de objetos concurrentes sufren ambos tipos de indeterminismo. El indeterminismo global, debido al tiempo en atender cada petición e indeterminismo local, debido a la estrategia usada para ir seleccionado tareas de la cola privada del objeto. El modelo de objetos *Parallel Objects* (PO), es un modelo donde el paralelismo y el indeterminismo pueden ser expresados con facilidad. La llegada de los clusters al campo de la HPC como plataforma de computación paralela, robusta y barata provocó que el modelo de programación usado se apoyase en la programación estructurada complementada con el paso de mensajes como elección para adoptar un modelo de computación muy cómo para toda la comunidad de desarrolladores.

Las nuevas arquitecturas están promoviendo un modelo capaz de conciliar ambas perspectivas. Sin embargo, hay artículos y numerosos trabajos en los que se apuesta fuertemente en la construcción de software de altas prestaciones en el que se usen composiciones de objetos (tambien denominadas High Level Parallel Compositions o HLPC) en las que se proporcionan schedulers internos y mecanismos de control de concurrencia. Estas construcciones o HLPC también son conocidas en la literatura como CPAN (acrónimo en castellano) [188] (que en este caso se apoyan en librerías desarrolladas en C++ y pthreads).

Tal y como se citó antes (capitulo 1), se busca un tipo de paralelismo de grano más fino, en este sentido se desarrolla una nueva versión de BICAV. Los algoritmos de reconstrucción paralelos suelen adoptar la filosofía *maestro-esclavo* donde un procesador se dedica a repartir entre el resto de procesadores los *sinogramas*. Cada procesador que recibe un conjunto de sinogramas trabaja sobre ellos realizando el cálculo de la proyección y enviando esta información al procesador maestro. El procesador maestro se encarga de recuperar las proyecciones parciales, comparar con la proyección experimental y transmitir las diferencias a cada procesador para que corrija el volumen parcial que ha de reconstruir.

El grano del paralelismo aplicado en esta estrategia paralela es grueso. Adoptar otras formas de trabajo podría facilitar la incorporación de un grano de paralelismo más fino. La decisión de abordar la reconstrucción usando un grano más fino, además de venir justificada en el capitulo primero, tiene su razón en que estos algoritmos suelen comportarse, en términos de eficiencia, en máquinas con muchos procesadores (no necesariamente potentes en cálculo) y conectados en una topología de malla. Un cluster de estaciones de trabajo puede ajustarse muy bien a este modelo de máquinas, pues si bien sus procesadores son rápidos, cuando se contrasta con la latencia de red se puede ver al sistema como una máquina con procesadores lentos conectados en malla. El comportamiento de estos algoritmos también mejora sensiblemente la ejecución del problema en un procesador individual.

Partiendo de la implementación base de la versión MPI del programa de reconstrucción y para permitir que esta fuera una implementación lo más independiente de la arquitectura posible, que pueda ser ejecutado en un supercomputador, en un multiprocesador, en cualquier arquitectura paralela es importante que el modelo de programación ayude a esconder o abstraer la arquitectura en la medida de lo posible.

En la implementación se ha escogido al lenguaje de programación *Charm++* [144] por que

reúne dos requisitos fundamentales, el primero es que se apoya sobre el *runtime* que durante todo este trabajo hemos considerado como idóneo para nuestro marco de trabajo. El segundo motivo es que es un lenguaje paralelo orientado a objetos cuyo modelo de ejecución depende del envío de *mensajes* entre los mismos. En este tipo de lenguajes paralelos, la ejecución, invocación de métodos y la sincronización se llevan a cabo enviando mensajes entre diferentes procesos. Que Charm++ sea orientado a objetos nos proporciona flexibilidad a la hora de codificar el problema.

Quizás la característica más atractiva de este esquema de implementación sea su esquema de envío de mensajes con el que la creación de una aplicación distribuida y asíncrona es muy sencillo, permitiendo a su vez que la aplicación pueda escalar facilmente cualquier número de procesadores.

La implementación presentada en este apartado por tanto, se apoya en el uso de un modelo de objetos concurrentes ofrecidos por un framework que abstrae al programador de la arquitectura subyacente. El citado runtime ofrece, por separado, un planificador o scheduler para cualquier aplicación que se ejecute bajo su amparo. De este modo se evitan indeterminismos locales porque el planificador es parte del propio runtime (RTS).

En este caso particular se ha implementado un High Level Parallel Composition (usando la librería de hebras ofrecida por el framework, quickthreads). Este HLPC se implementó siguiendo el patrón de desarrollo *farm*[188], donde existen un grupo de objetos concurrentes, capaces de trabajar en paralelo bajo las directrices de un objeto maestro (ver figura 3.22). El modelo seleccionado para controlar la concurrencia dentro de este HLPC ha sido MAXPAR [189]. Los resultados que se han obtenido como consecuencia de esta implementación animan a explorar en esta dirección el desarrollo de aplicaciones científicas y su despliegue en arquitecturas dispuestas para la computación de altas prestaciones. En el modelo propuesto cabe destacar la existencia del planificador (ofrecido por el propio runtime, útil para la cola de mensajes, y para difuminar los efectos negativos de los indeterminismos global y local). El uso de la librería de hebras *quickthreads* ofrece un medio eficaz para ofrecer a la implementación la habilidad de realizar cambios de contexto agilmente, cuando sea necesario y siempre en aras de favorecer la concurrencia y mantener el progreso de la computación.

La figura 3.22 describe la arquitectura de la solución propuesta. La entidad *manager* la proporciona el framework de Charm++. El objeto etiquetado como *main* es el único responsable

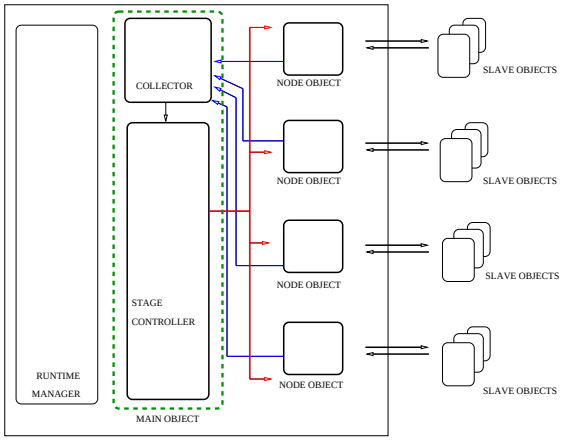


Figura 3.22: Modelo de Construcción Paralela de Alto Nivel que se ha usado para la versión de grano fino.

del avance del programa. Los objetos *nodo* son objetos que se emplazan en cada procesador o núcleo y actúan de intermediarios entre el objeto main y los esclavos de cada procesador. Los esclavos son objetos concurrentes que se limitan a realizar las tareas de reconstrucción, independientemente del procesador en el que se localicen.

3.6.1. Implementación

Las aplicaciones objeto de este estudio son aplicaciones paralelas. Aunque en las arquitecturas actuales *concurrencia* y *paralelismo* se encuentran relacionados[190], la concurrencia puede mejorar el rendimiento en tres formas fundamentalmente:

- Puede ocultar la latencia (aspecto explotado en la sección 3.5.1).
- Puede reducir la latencia.
- Puede incrementar el rendimiento del sistema, si se es capaz de dividir la computación en componentes autónomos.

Usar la concurrencia para ocultar la latencia es una técnica íntimamente ligada a la naturaleza del problema, además esta tarea precisa de un algoritmo paralelo. Usar la concurrencia para reducir la latencia requiere cargas de trabajo lo suficientemente importantes como para suplir los costes de coordinar diversos elementos computacionales, este aspecto también depende del problema. Cuando los problemas se resisten a ser paralelizados o no tienen una latencia apreciable que ocultar, la tercera forma en la que la ejecución concurrente puede mejorar el rendimiento

es incrementando el desempeño del sistema. En este caso, en lugar de usar el paralelismo para acelerar una operación individual, se acude a la ejecución concurrente de múltiples componentes con lógica secuencial, de este modo es posible dar cabida a más trabajo simultáneamente. Es importante notar que un sistema que se apoye en la concurrencia no tiene por qué ser multihebrado. Es más, estos citados componentes caracterizados por, entre otras cosas, no compartir su estado (objetos) pueden incluso ser secuenciales. La forma de habilitar, en este escenario, la *compartición* puede encontrarse en el desarrollo de componentes especializados en trabajar paralelamente con estados compartidos, idealmente estos componentes serán exclusivamente aquellos capaces de operar con corrección en situaciones donde la concurrencia sea crítica.

La adaptación de aplicaciones científicas hacia estos nuevos entornos debería realizarse con el máximo de transparencia. Esto debe ser así porque de otro modo nos enfrentamos bien a la reescritura completa de la misma o a una refactorización donde se debe asegurar que la aplicación no sufra alteraciones. En esta dirección se debería explotar al máximo cualquier técnica de paralelización automática y se deberían, también, considerar los paradigmas de programación en memoria compartida en función de su facilidad de uso, rendimiento y habilidad para dar soporte a diversas tareas de programación. Sin embargo, el hecho de migrar aplicaciones donde el paralelismo deba ser incluido de manera explícita es inevitable, por lo que la migración debería apoyarse en la habilidad de conseguir *expresividad* a través de las herramientas usadas.

Por un lado, la necesidad de trabajar sobre *plataformas concurrentes*[174] y por otro lado, el requisito de realizar las portaciones entre plataformas de la manera más transparente exigiendo cada vez más expresividad[140], hace que *frameworks* como *Charm++*[144] puedan ser considerados una seria alternativa.

En contra de lo que se podría llegar a pensar hasta hace un par de años, ahora, si se desea realizar la adaptación con cierta comodidad y sin artificios se requiere disponer de uno o varios modelos de programación de alto nivel (la integración de paradigmas puede llegar a ser, incluso, una potente alternativa). La necesidad de estos modelos de alto nivel viene relacionada con ayudar a la adaptación a través de la expresividad del lenguaje empleado. La paralelización automática es un área de investigación en la que existen interesantes avances, prometedores pero por ahora insuficientes. Las aplicaciones actuales se paralelizan manualmente, normalmente a través de APIs (como es el caso de MPI). En el caso de los sistemas de memoria compartida, las *herramientas o técnicas* más usadas son las *pthreads* y *OpenMP*. Mientras que el uso de

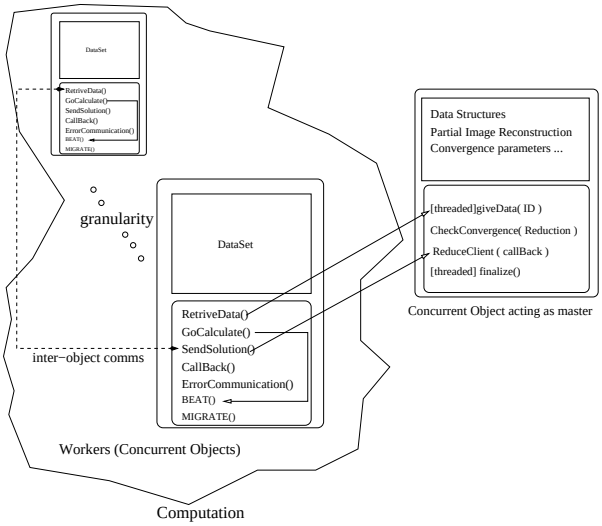


Figura 3.23: Implementación Orientada a Objetos

pthreads supone reorganizar toda la estructura del programa (cuidadosamente), la inserción de directivas *OpenMP* supone adaptar el código de manera inmediata. Sin embargo, las últimas versiones de *OpenMP* aún distan de proporcionar la expresividad de localidad y modularidad que serían deseables en aplicaciones que vayan a ser *desplegadas* en arquitecturas multicore [140].

Charm++, es un framework que amplía al modelo de programación ofrecido por C++. Las principales aportaciones de Charm++ son el uso de objetos concurrentes y de un modelo de ejecución asíncrono basado en el paso de mensajes entre objetos (acceso a servicios). Estas han sido las dos razones principales por las que se ha usado Charm++ como modelo de programación (ver Figura 3.24), no obstante existen muchas otras que también justifican su uso, como el disponer de una capa de software que abstrae de la arquitectura, planificadores integrados, API para la instrumentación de la actividad de cada objeto, etc. El modelo de objetos de Charm++ se basa en entidades de computación basadas conceptualmente en el modelo de objetos activos (active object model) [191]. La portación de la aplicación de reconstrucción, para su ejecución en la plataforma multicore, ha consistido en conservar la naturaleza iterativa del problema. La implementación ha requerido aplicar cambios a la arquitectura del problema para acomodar el concepto de orientación a objeto.

El modelo implementado requiere de objetos capaces de controlar la reconstrucción siguiendo la metodología *cliente–servidor o maestro–esclavo*. Estos objetos que dirigen la computación se encargan de tareas administrativas tales como crear las estructuras de datos necesarias, iniciar la

instanciación de todos los objetos esclavos, control de convergencia hacia la solución, etc.... Los objetos encargados de las tareas de cálculo (objetos esclavos) obtienenen las instrucciones desde el objeto maestro. La forma de solicitar instruciones o datos es diferente al modelo convencional (MPI), en este caso son los esclavos los que solicitan al maestro (que se compone de métodos concurrentes y atómicos o apropiativos) tanto el inicio de una nueva fase como los datos con los que deben trabajar. De este modo la comunicación no se lleva a cabo de uno a muchos sino de uno a uno, existiendo la posibilidad de encontrar en un instante de tiempo dado varias comunicaciones simultáneas.

De modo que los objetos deben solicitar servicios (o invocar métodos) del objeto master (master) para obtener los datos que les son necesarios para progresar con la computación. Un aspecto interesante, motivo de estudio es el método para comunicar la información (si el objeto maestro proporciona todos los datos necesarios para el objeto de una vez o únicamente los que les sean necesarios para esa iteración). En cualquier caso, toda comunicación se lleva a cabo a través de invocación de métodos (ya se esté trabajando en la plataforma multicore o en la plataforma cluster) en contraste con la implementación MPI que emplea paso de mensajes.

Tal como se indicó anteriormente, esta implementación de grano fino considera la creación de un *objeto maestro* o *master* y de un conjunto de *objetos esclavos* subordinados a un grupo de objetos llamado *objetos de grupo* (ver figura 3.22). El objeto maestro es el responsable de crear al resto de objetos, entre ellos, de crear a un objeto grupo por cada procesador (o núcleo), a la par que se van creando objetos (y sin esperar a que todos estén creados) se inicia la solicitud de creación de *esclavos*. Cada objeto grupo solicita al objeto master los datos que sus esclavos van a necesitar. Tan pronto lleguen los datos, los esclavos podrán iniciar su trabajo. Toda comunicación entre esclavos se hace a través de la invocación de métodos. La reconstrucción, por tanto, en esta versión es *guiada por mensajes (invocación de métodos)*. En un escenario convencional la *conversación* entre el objeto *master o maestro* y los objetos *esclavos*, para una iteración, se describe esquemáticamente en el siguiente listado de código (y en la Figura 3.23). La comunicación entre objetos en procesadores físicos diferentes (comunicación remota) se permite a través de objetos especiales denominados *proxies* que actúan como representantes locales de los objetos remotos.

El objeto *maestro* crea a todos los *esclavos*. Tras realizar todas las tareas de inicialización necesarias, el objeto *maestro* invoca a cada objeto grupo.

```
gnodo   = CProxy_Grupo::ckNew();
```

```
CkCallback *cb = new CkCallback(
CkIndex_Main::composicion(NULL),   mainProxy);

gnodo.ckSetReductionClient(cb);
```

De esta forma se solicita la creación de los *objetos de grupo*. En el momento de la creación, el *objeto maestro* le pasa un *callback* al grupo con el fin de recoger más adelante las reconstrucciones parciales. Tan pronto hayan sido creados los objetos *de grupo* y *esclavos* se hace preciso solicitar los datos sobre los que cada esclavo ha de trabajar. Cada *esclavo* por tanto, solicita de manera concurrente al *maestro* el envío de los slabs.

```
/*From the group object the data is requested.
A message is built where the ID for this object
is stored, in order to receive the appropriate
slab. mE is a message object.In this line the
group object uses its proxy object to contact
the master object and invoke the data request
service */

mainProxy.giveData(mE);
```

El objeto *maestro* reparte los datos (slabs) solicitados a cada objeto *grupo*

```
/*The main object packs into a message the
requested data (slab)*/

for (f=start; f<end;f++)
{
msg->Hpart[index]=sinogram[f];
index++;
}

/*And delivers the buffer. */
```

```
gnodo[rank].retrieveData(msg);
```

Los *esclavos* trabajan con la información que le llega a través de los objetos *grupo*.

```
/*At the buffer arrival, the work starts*/

Grupo::getSinograms(matrixHpart *msg)
{
int i;
int rank = CkMyPe();

/* Make space for data at the object. And
fast-copy it */

YP=(float*)malloc(sizeof(float)*msg->pixels);
memcpy(YP,msg->Hpart,sizeof(float)*msg->pixels);

/* Do the work : this is equivalent to the
stripped part in Figure 1*/

GoCalculate(HP, YP, fpart, numf);
}
```

Tras cada iteración, cada objeto debe contribuir con la reconstrucción parcial de manera similar a esta:

```
contribute(bytes,fpart,CkReduction::sum_float);
```

El objeto *maestro* entonces comprueba los criterios de convergencia (tras realizar la recomposición). El programa termina al satisfacerse las condiciones de convergencia.

El cometido de cada objeto esclavo es realizar, iterativamente, la reconstrucción parcial de la porción de volumen que le haya sido encomendada. Varios objetos pueden estar activos a la vez en el mismo procesador (o núcleo) trabajando de manera concurrente y coordinada, explotando las condiciones de concurrencia puede acelerarse la convergencia a la solución. La solución orientada a objetos también puede dotarse de mecanismos de balanceo de carga, idénticos a los que se aplican a la solución de grano grueso. La ventaja de esta implementación es que se puede explotar el grano fino para incluir un objeto que decida si activar o no la instrumentación para alterar la

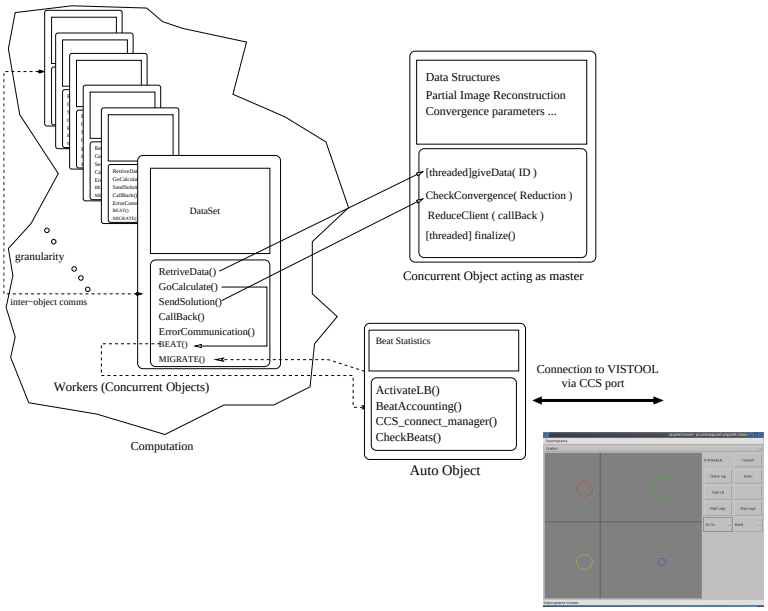


Figura 3.24: Implementación adaptativa

asignación objeto-procesador (o núcleo). De este modo se puede dotar de cierta *adaptatividad* a la aplicación de grano fino, mucho más versatil que la que se puede incluir en la de grano grueso (este aspecto se desarrolla en el capítulo siguiente). La citada adaptatividad, por tanto, se puede conseguir a través de un objeto, que no forma parte del dominio del problema de la reconstrucción pero que auxilia en conseguir mayor escalabilidad y rendimiento (ver *el objeto auto*) en la Figura 3.24. Este objeto dirige la intervención del balanceador (no su política). Este objeto tiene también el cometido de informar, a través de una serie de directivas insertas en el código de sus métodos sobre el rendimiento de cada objeto esclavo, sin necesidad de recurrir a la instrumentación del framework Charm++. De modo que se pueda visualizar (durante la ejecución o al término de esta) detalles de la ejecución de la aplicación. Para tal efecto se ha desarrollado una aplicación que recoge las intrucciones enviadas desde este objeto y las interpreta. La aplicación visual (y como la insercción de directivas en el archivo o a través del puerto CCS puede haberse alterado dado que diversos objetos pueden haber informado a la vez y la secuencialidad de la escritura no se garantiza) posee un analizador que permite corregir potenciales errores *sintácticos* incluidos en la información de rendimiento. Esta herramienta visual puede ser usada para monitorizar la actividad del objeto auto o para decidir la intervención manual del balanceador en la aplicación.

Se han realizado experimentos en una plataforma multicore (Intel Core 2 Quad Q6600 que cuenta con 8M Cache, cada núcleo trabaja a una frecuencia de 2.40 GHz, y el bus frontal

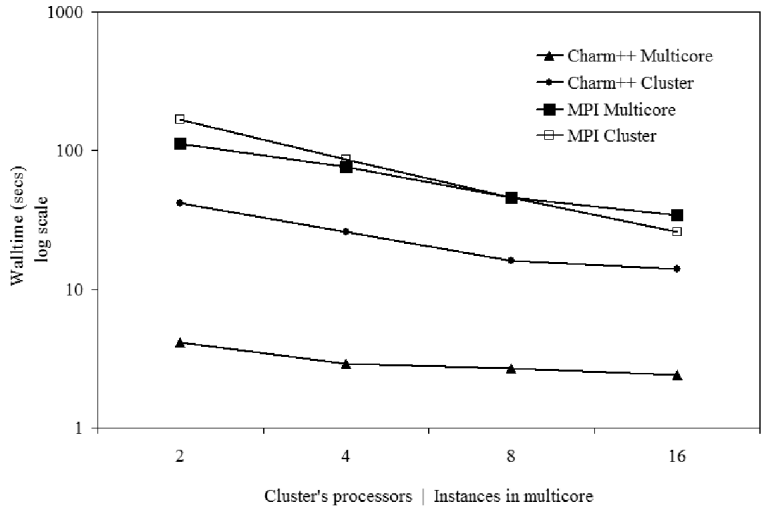


Figura 3.25: Ejecución en cluster y en multicore.

o FSB a 1066 MHz) y en el cluster VERMEER. El propósito de los experimentos ha sido además de comprobar la fidelidad en la reconstrucción, comprobar que el problema escala de manera más adecuada que en la versión MPI. Desde el punto de vista que las operaciones aritméticas son idénticas en ambos casos (MPI y Charm) la diferencia se puede encontrar en la optimización empleada por el compilador usado por MPI y el usado por Charm++. En ambos casos mpicc y charmc -compiladores para la versión MPI y para la versión Charm++- son *wrappers/empaquetadores* de gcc/g++. Por tanto, se puede pensar que las diferencias en ambas versiones se debe exclusivamente al empleo de los diferentes modelos de programación.

La Figura 3.25 muestra el comportamiento de la reconstrucción cuando se usa la aplicación desarrollada con MPI y la desarrollada sobre el framework de Charm++, tanto en un cluster como en un procesador multicore. El eje de abscisas representa el número de procesadores si se usa el cluster y el número de instancias (tanto de procesos MPI como de objetos) por core, si se usa el procesador multicore. En este caso, la reconstrucción se aplica sobre un conjunto de datos bastante reducido (reconstrucción 2D de una imagen) tanto en el cluster como en el multicore y para ambas versiones, MPI y Charm++. Los tests realizados con la versión orientada a objetos (Charm++) han sido lanzados siempre con cuatro objetos por procesador en el cluster.

Tal y como muestra la figura 3.25, la versión de Charm++ muestra un mejor comportamiento (reducir la granularidad permite mejorescalabilidad [133]) que la versión implementada usando C y MPI. En el caso de estar usando un cluster, el proceso de reconstrucción (versión) MPI

Tabla 3.8: Walltime experimentado en un cluster.

Cluster		
Procs	MPI WallTime	Objects WallTime
2	366	42
4	86	26
8	46	16
16	26	14
32	n/p	n/p

Tabla 3.9: Walltime experimentado en un multicore.

Multicore		
Objects	MPI WallTime	Objects WallTime
2	111,449	4,1
4	75,651	2,891
8	45,496	2,697
16	34,136	2,4
32	33,25	2,121

es el único flujo de control activo en cada procesador, el Front Side Bus (FSB) se está usando unicamente para servir las peticiones de acceso a memoria del proceso, la caché está siendo usada exclusivamente por el proceso MPI y por tanto el patrón de accesos del mismo se ve favorecido. No obstante el principal motivo de pérdida de rendimiento en este escenario es el uso de la red y sus latencias asociadas. Comunicar dos procesos MPI a través de la red es costoso, incluso con los chipsets actuales que están extremadamente optimizados se emplean miles de ciclos de reloj esperando a que se completen las etapas de comunicaciones. Es un hecho que en las plataformas cluster las comunicaciones no bloqueantes o alternativas como las presentadas en [141] se emplean para intentar reducir el impacto de las latencias. Desarrollar las aplicaciones usando las metodologías orientadas a objetos (donde los objetos cumplen ciertas condiciones como las que ofrece el framework Charm++) puede ser considerado como una alternativa capaz de reducir el impacto de las latencias. La Figura 3.25 muestra cómo en el entorno del cluster, la versión orientada a objetos (Charm++) se comporta mejor que la version MPI al afrontar las tareas de reconstrucción (estos resultados son una extensión de los resultados vistos en los experimentos de grano grueso), la granularidad y la concurrencia están siendo de bastante

ayuda no son la única razón de la mejora experimentada. La orientación a objetos incorpora características como la encapsulación que mejora el uso de las cachés (aspecto que redunda en las ganancias de rendimiento)[192],[185].

Las experiencias realizadas sobre la arquitectura multicore (Intel Core 2 Quad Q6600) muestra que la versión desarrollada con MPI alcanza un punto (8 procesos, dos procesos por núcleo) donde la contención de la caché (compartida por cada par de núcleos) afecta al rendimiento. Además MPI se basa en el paso de mensajes y aunque la red permanece intacta en estas experiencias, cuando se comparte memoria el paso de mensajes se traduce en una copia desde la memoria privada del proceso que realiza el envío al área de memoria compartida, este proceso de copia al realizar la escritura en un área compartida no está exento de conflictos que pueden ser causados por otros núcleos. El hecho de que la versión MPI para el procesador multicore no se comporte mejor que cuando fue lanzada en el cluster se debe a que la frecuencia de reloj de cada núcleo en el procesador multicore es bastante inferior (casi la mitad) que en los procesadores del cluster. A esto se le añade que, atendiendo a políticas de bajo consumo, los núcleos limiten su frecuencia de trabajo. Sin embargo, la versión orientada a objetos contempla optimizaciones de las que no disfruta la versión MPI, como por ejemplo, el mejor uso de la caché debido a la encapsulación, el modelo de comunicación basado en invocación asíncrona de métodos en el que si el método invocado puede contemplar cierta atomicidad, aspecto este que puede asemejarse, en relación con el acceso a memoria, con políticas transaccionales elementales de acceso a memoria. Además de lo citado, cabe destacar que con el uso de hebras de usuario (quickthreads) los cambios de contexto se realizan con bastante agilidad lo que hace que esta versión sea considerada como una seria competidora de la versión MPI.

El uso de objetos permite una mejor descomposición del problema, de acuerdo al trabajo realizado en [133] se muestra que reduciendo la granularidad se consigue mejorar el rendimiento y la escalabilidad, pero siguen apoyándose en el modelo de proceso. De este modo el beneficio de poder explotar la concurrencia no es tan alto como cuando se emplean objetos. Concretamente, dentro de este trabajo indican que incrementar la granularidad permite tener *más procesos* y acometer antes las tareas. Pero el número de procesos concurrentes, en el caso que se muestra en sus experimentos no es alto. De aquí, y por comparación directa, se derivan dos consecuencias : *el cambio de contexto puede ser costoso*, *se precisa, igual que en nuestro caso, cargar en memoria el RTS (IOS –Internet Operating System– más todas las librerías de gestión desarrolladas.* En

[193] se pueden encontrar detalles de las optimizaciones que hacen que el runtime Charm sea una opción a considerar en las plataformas multicore.

3.7. Línea de trabajo abierta: implementación propia de una arquitectura de virtualización

Esta sección presenta el primer entorno que se desarrolló en aras de dar servicio a las aplicaciones de ciencia e ingeniería según los aspectos mencionados en la lista anterior. El entorno se desarrolló usando Java. Java es tanto un lenguaje de programación como una máquina virtual que abstrae al procesador nativo. Desde este punto de vista Java ya proporciona bastante base sobre el entorno que precisamos para portar nuestras aplicaciones o incluso para diseñar las nuevas y que disfruten de la deseada abstracción.

En primera instancia decidimos implementar una extensión sobre la máquina virtual Java de Sun Microsystems. De este modo pretendimos aprovechar que la JVM (o Java Virtual Machine) ya ofrecía un modelo de objetos y una potente librería de gestión de hebras (*green threads*). Al disponer de gran parte de la infrastructura, sólo restaba implementar una extensión a la máquina virtual que nos permitiera sobre todo abstraer el *concepto de proceso*, de este modo sería posible adaptar aplicaciones ya desarrolladas al nuevo modelo orientado a objetos. A esta extensión de la JVM también sería preciso dotarla de librerías de gestión donde se pudiera decidir la politica de sincronización, de cancelación, etc ... de este nuevo concepto abstracto (proceso). En una etapa final y conseguidos estos dos pasos inciales tendría que habilitarse un sistema global de direcciones para permitir la colaboración entre varias máquinas virtuales, dando la impresión de que todas son la misma (ya existe un trabajo parecido, aunque no abordan el concepto de proceso abstracto) [194].

Los detalles más significativos del entorno desarrollado son :

- El uso de la abstracción de un Proceso, que embute a una aplicación Java.
- El uso de la abstracción de Hebra, capaz de embutir a su vez a un proceso.
- El uso de un runtime (sobre la máquina virtual), llamado Servidor, que gestiona la creación de Procesos y su ejecución apropiativa dentro de hebras de usuario (green-threads) implementadas con el modelo de Java. La gestión de este servidor se extendió con librerías que

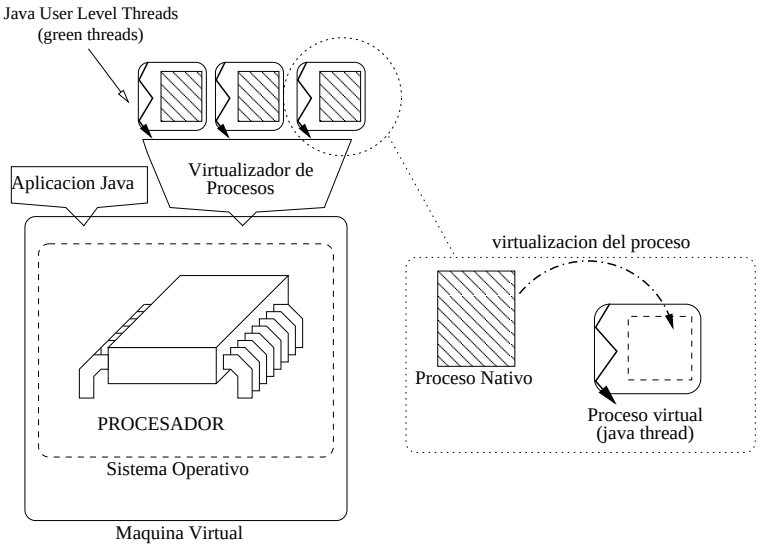


Figura 3.26: Esquema del RunTime Java

gestionan la convicencia de las hebras que abstraen a los *procesos*.

Como puede verse en la figura 3.26 la implementaciíon contempla la incorporación de cada proceso en el que se descompone la aplicación dentro de una clase Java llamada *proceso* que hace las veces de proceso virtual dentro de la máquina virtual Java. Esta clase proceso debe estar embebida en una hebra, ver figura 3.27. De este modo se pueden mantener en ejecución (dentro de un conjunto de hebras de usuario) todos los procesos que deseemos (con el límite lógico impuesto por la memoria asignada al runtime Java). El Runtime implementado actúa como un servidor de aplicaciones, donde cada aplicación es cada uno de los procesos de la aplicación paralela. El siguiente listado recoge, como ejemplo, la virtualización de procesos del *runtime Java* diseñado.

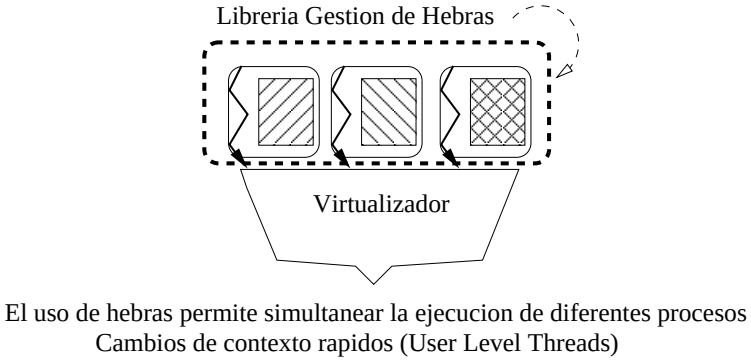


Figura 3.27: Hebras y Procesos virtuales dentro del runtime

```
import java.lang.* ;
import java.lang.reflect.Method;

public class Servidor
{
   public static void main(String []arg)
   {
     System.out.println("Iniciando la clase Servidor \n");
     iniciarEnHebra("Aplicacion1",arg);
     iniciarEnHebra("Aplicacion2",arg);
   }

   public static Proceso iniciarEnHebra(String n,String[] arg)
   {
      Proceso proceso = new Proceso(n, arg);
      //***************  Nuevo: **********************
      ThreadGroup grupo = new ThreadGroup("main");
      Thread hebra = new Thread(grupo, proceso);
      //***************  Nuevo: **********************
      hebra.start();
      return proceso;
   }
 }
```

```
class Proceso implements Runnable{
  String n;
  String[] Argumentos;
  //***************  Nuevo: **********************
  ThreadGroup hebraEnCurso;
  //***************  Nuevo: **********************

  public Proceso (String nombre, String[] arg)
  {
    n = nombre;
    Argumentos  = arg;
```

```
}

public static void iniciarOtraClase(String n, String[] arg)
throws Exception
{
   //Obtener la clase
   Class objeto = Class.forName(n);
   //Localizar el metodo main(String []) de esa clase
   Class[] tipoParametrosMain = {args.getClass()};
   Method main=objeto.getMethod("main", tipoParametrosMain);
   //array con los argumentos a pasarle a la clase
   Object[] parametrosMain = {arg};
   //Iniciar la clase
   main.invoke(null, parametrosMain);
 }

public void run()
{
   //***************  Nuevo: **********************
    hebraEnCurso=Thread.currentThread().getThreadGroup();
   //***************  Nuevo: **********************

   try{
      iniciarOtraClase(n, Argumentos);
      }catch(Exception e){
         System.out.println(" Excepcion ... ");
         e.printStackTrace();
      }
}

public void terminar()
{
  if(hebraEnCurso != null)
         hebraEnCurso.stop();
}
```

```
}
```

La implementación del entorno de trabajo debería ser capaz de resolver estos inconvenientes, el uso de una máquina virtual tiene la gran ventaja de proveer una *homogeneización* conveniente a la hora de mover unidades de cómputo entre procesadores, aunque su arquitectura sea diferente.

La arquitectura de este entorno software es, sin duda y como atestigua la Figura 3.28, la que precisamos para la portación de aplicaciones ya desarrolladas y la que puede facilitar el implementar aplicaciones científicas desde cero sin necesidad de tener que conocer demasiados detalles relevantes de la arquitectura hardware subyacente gracias a la existencia de la abstracción provista por el *runtime*, tal y como se aconseja en [180]. Por otro lado el precio de la ubicuidad es alto porque en este entorno el *runtime* actúa como interprete. En la Figura 3.28 es puede ver el efecto de la virtualización. La imagen de la izquierda muestra a los dos procesos JAVA ejecutados concurrentemente, embebidos en las hebras (green threads) de usuario que proporciona la máquina virtual Java. La imagen de la derecha muestra a los dos procesos ejecutados en el procesador, donde la ejecución de ambos no se realiza concurrentemente. En este caso la concurrencia se ha alcanzado de manera directa, sin incluir ningúna línea de código en los procesos ejecutados. Este es el principal activo de usar tal implementación. Ante problemas de mayor envergadura, los procesos a ejecutar deberán proteger convenientemente sus *variables globales*.

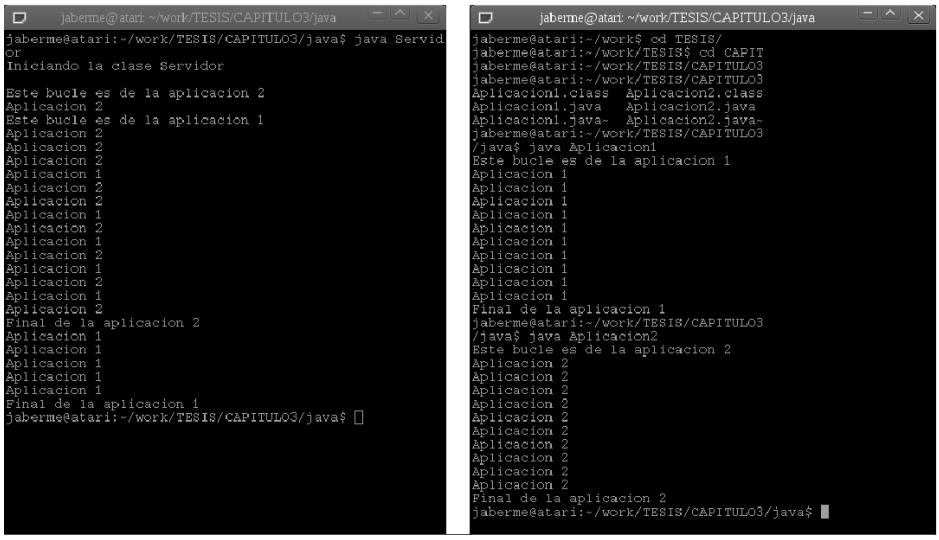


Figura 3.28: Virtualización de procesos en Java

A pesar de su ideonidad, el desarrollo de este sistema presenta algunas dificultades, abordarlas es un propósito excesivamente ambicioso para este trabajo, de entre ellas, las más significativas son

- Se precisa una implementación muy cuidada de la gestión y arbitraje de las hebras que gestionan los procesos virtuales.
- El runtime debería ser capaz de instrumentar la actividad de cada hebra para detectar el momento exacto en el que ha de conmutar la ejecución de una por otra.
- Las comunicaciones entre procesos virtuales se realizan a través de llamadas a procedimientos remotos, sin distinguir si las hebras comunicantes están o no en el mismo procesador. Se precisa un sistema de comunicación ofrecido por el propio runtime, en su lugar.
- Si se incorporan procesos que usan las librerías MPI, la señalización de las mismas colisiona con la señalización existente entre la máquina virtual Java y el sistema operativo.

La implementación presentada es correcta y funcional únicamente para el caso de aplicaciones desarrolladas en Java pues estas aplicaciones ya contemplan que bajo ellas habrá una máquina virtual. Cuando se pretendió llevar aplicaciones C bajo este entorno, experimentamos problemas relativos a la *señalización* entre los procesos y la máquina virtual Java. La solución a esta serie de inconvenientes pasaba por añadir a la implementación una extensa librería que manejara las posibles llamadas que hace C al sistema (e incluso a otras librerías como MPI). En suma, se debería haber abordado el desarrollo de un *runtime* únicamente, sin usar ninguna *máquina virtual*.

Fue el descubrir soluciones similares y en fase de experimentación por otros grupos de investigación, con resultados iniciales esperanzadores lo que nos hizo centrarnos en este nuevo entorno explicado a continuación. El desarrollo de un marco de trabajo basado en máquinas virtuales de alto rendimiento, es actualmente, una línea de trabajo que tenemos abierta.

3.8. Conclusiones

Este capítulo ha presentado los resultados, más importantes, obtenidos al asumir el modelo hebrado como modelo de cómputo en aplicaciones científicas dentro del ámbito de la reconstrucción 3D de imágenes. Usar las técnicas de programación multihebra como mecanismo de

conmutación rápida y la orientación a objetos como mecanismo de modelado y lenguaje paralelo, ha demostrado ser un sistema útil a la hora de incrementar el rendimiento de la aplicación a partir de la concurrencia. Otro beneficio directo es que no se hace preciso implementar técnicas que ayuden a ocultar las operaciones de comunicación ya que aunar multihebra y orientación a objetos lo proporcionan de manera directa.

El estudio abordado en este capítulo se ha basado principalmente en el uso de la capa AMPI. AMPI se apoya en un potente runtime llamado Charm++ que favorece la conmutación entre hebras de usuario cuando la cpu deja de registrar actividad en sus ciclos. A través de experimentos se ha ilustrado como BICAV, nuestra aplicación de reconstrucción de imágenes obtiene un beneficio directo ya que la diferencia en el tiempo de CPU frente al tiempo WallTime se ha conseguido acercar a cero. En contraste a lo ocurrido con la aplicación MPI en la que al incrementar el número de procesadores se acentúa esta diferencia, acusando aún más el efecto de la latencia.

Durante este estudio se han variado parámetros entre los que se cuentan el número de procesadores físicos y el número de procesos virtuales (hebras) pudiendo constatar que la versión hebrada de BICAV ha experimentado mejor escalado que la versión MPI. De estos experimentos podemos concluir que las hebras de usuario son una interesante y eficaz alternativa para la programación de aplicaciones científicas y paralelas, aplicaciones que como hemos visto, pueden aprovechar el tener varios flujos de control por procesador.

En el caso concreto de nuestra aplicación, BICAV, esto significa que es posible incrementar el número de bloques a usar, comparado con la versión MPI. Incrementar el número de bloques supone acelerar la convergencia del algoritmo (incrementando también las comunicaciones necesarias) sin que el rendimiento de la aplicación pueda verse afectado.

Es preciso mencionar que en los experimentos realizados se ha desactivado la tecnología HT en los procesadores. La comparación de las dos implementaciones ha sido llevada a cabo en condiciones de igualdad para ambas versiones. Cabe citar en relación con esto que el rendimiento de la versión AMPI se ve beneficiado de haber tenido HT activado o en el caso de haber recurrido a la plataforma multicore.

Estos resultados lejos de ser un condicionante concluyente, invitan a buscar una opción más que además de permitir el escalado en arquitecturas paralelas, permitan un escalado adaptativo, es decir buscando siempre la mejor distribución de hebras. Aspecto que da pie a la lectura del

siguiente capitulo donde se han explotado diferentes técnicas y algoritmos.

El desarrollo de programas paralelos puede llegar a ser bastante tedioso para determinadas aplicaciones cientfícas. Las técnicas de programación paralela tradicionales eran además propensas a hacer que el programador hiciera su trabajo en estrecha relación con la plataforma sobre la que se iba a poner el desarrollo en producción de modo que gran parte del esfuerzo se dedica a detalles relacionados con las arquitecturas, dejando en segundo plano detalles relacionados con el ámbito del problema. Abstraer el desarrollo de la plataforma puede ser beneficioso y además productivo, en este sentido se pueden usar técnicas de programación de alto nivel basadas en *abstracciones*. En este capítulo se han presentado abstracciones a dos niveles diferentes. En un primer lugar se ha presentado una técnica consistente en *embeber* procesos tradicionales en complejas estructuras compuestas por objetos y hebras de usuario. Esta técnica, de grano grueso, ha mostrado ser bastante eficiente en las plataformas cluster donde con determinadas transformaciones en el código, es posible obtener ganancias sustanciales en el rendimiento. En segunda instancia se ha presentado una técnica consistente en el desarrollo de la aplicación, partiendo desde cero, usando la metodología orientada a objetos, donde los objetos cumplían ciertas condiciones. En este caso, la versión obtenida ha demostrado mejor comportamiento tanto en el cluster como en arquitecturas multicore. La razón para este último caso es que se usan abstracciones que sin contemplar la arquitectura subyacente[150], [195], son capaces de modelar el paralelismo y la concurrencia necesarias con naturalidad y desde las mismas construcciones del lenguaje usado.

En el desarrollo de la segunda implementación se han usado construcciones de alto nivel para modelar el comportamiento del sistema, dentro de estas construcciones cabe citar a MAXPAR como modelo de control de concurrencia.

Se debe volver a indicar que la versión de grano grueso no permite modelar el paralelismo de manera natural, pues se emplean objetos *predefinidos*, llamados wrappers, para embutir a un proceso, los servicios que este objeto ofrece como métodos invocables dependen de la implementación MPI, pues se sustituyen las funciones de paso de mensajes por métodos en estos objetos *empaquetadores*. De modo que las ganancias, no se adivinan tan sustanciales como el caso de implementar toda la aplicación desde el principio.

Los resultados obtenidos experimentalmente, animan a explotar el camino que la orientación a objetos ofrece en el campo del HPC. Los resultados han determinado que usar objetos con-

currentes que siguen el patrón de los objetos *activos* y una apropiada plataforma de abstracción ayudan en mejorar las cotas de rendimiento y escalabilidad de la aplicación.

Capítulo 4

Estrategias de Equilibrado de Carga y Adaptabilidad

4.1. Introducción

Una de las características destacables de las aplicaciones científicas es la alta demanda de recursos computacionales de la que hacen gala. En la actualidad, el uso de la programación paralela y la existencia de plataformas paralelas tal como la plataforma cluster permiten que estas aplicaciones puedan ejecutarse en condiciones aceptables. Sin embargo y dada la naturaleza de un cluster (no es una máquina dedicada sino un aglomerado de máquinas interconectadas por una red de alta velocidad) especialmente si se emplea en un entorno de investigación, es probable que nuestra aplicación no sea la única que se encuentre en ejecución por lo que a la difícil tarea alojar recursos se le une la tarea de competir por ellos. Es por esto que incluso en aplicaciones totalmente regulares se pueden generar situaciones de desequilibrio debido a factores externos a la aplicación. Es evidente que el rendimiento de la aplicación que se ejecuta en el cluster se verá severamente afectado si no se tienen en cuenta los citados desequilibrios [1]. Es por esto que el balanceo dinámico de la carga se convierte en un factor importante en lo que al rendimiento se refiere. La distribución de las cargas computacionales de una aplicación determinada ha de ser tal que siempre se ejecute en las mejores circunstancias posibles, razón por la que se busca el balanceo dinámico y adaptativo, entendiendo por adaptativo al balanceador que a parte de carga también migra, de manera eficiente, la computación haciendo que ésta se ejecute en cada momento en el mapa óptimo de procesadores.

Seleccionar o elegir una estrategia de balanceo para aplicarla a una aplicación determinada, no es una tarea sencilla. La aplicación posee una serie de características que han de ser respetadas (e incluso conocidas) por la estrategia. Una estrategia de balanceo de carga depende estrechamente de la naturaleza de la aplicación y de la arquitectura hardware sobre la que esta se ejecuta (especialmente de aspectos como la topología de la red) [196], [197]. Por tanto, a la hora de diseñar una política de distribución de la computación en el cluster es preciso conocer muy bien todos los detalles relacionados con el *esquema de paralelismo empleado en la aplicación* (entre otras cosas, la Orientación a Objetos es de gran ayuda en este punto ya que permite revelar con cierto detalle aspectos sobre el paralelismo que pueden quedar ocultos a quien mantenga el código) y por otro lado es preciso tener constancia de los detalles técnicos que puedan afectar a la ejecución (ancho de banda de la red, arquitectura del procesador, rendimiento, etc ...).

Se ha citado que la aplicación y sus particularidades son determinantes a la hora de aplicar, con éxito, una estrategia de balanceo de carga. Una de las características más relevantes en este aspecto es la localidad de los datos, esto es, la dependencia que existe entre los datos. Aplicar una estrategia sin contemplar esta relación puede incrementar mucho más las comunicaciones en la red (por ejemplo) y hacer inútiles los esfuerzos vistos en el capítulo 3 para mantener oculta las latencias en las comunicaciones. Así pues, la localidad de los datos es una característica importante que condiciona a un gran porcentaje de aplicaciones científicas (de nuevo la orientación a objetos, ayuda a la hora de revelar las dependencias). Entre algunos ejemplos de aplicaciones donde se dan estas dependencias se cuentan aplicaciones de simulaciones físicas [198] (como fluidos, campos electromagnéticos, etc...) o aquellas relacionadas con el procesamiento de imágenes [199] como el filtrado, thinning, la segmentación o los algoritmos de reconstrucción 3D de imágenes tomográficas [200]. Diseñar una estrategia de balanceo para este tipo de aplicaciones implica considerar la distribución de datos de modo que la estrategia respete en lo posible esta distribución y asegurar que así se preserve la localidad de los datos. De no contemplar esta distribución, tras el primer intento de *remapear o rebalancear* la aplicación podremos experimentar un deterioro progresivo en el rendimiento debido a un incremento extra en las comunicaciones debido a la ruptura de la distribución impuesta por el algoritmo (de nuevo, la Orientación a Objetos asegura que la distribución de datos y las relaciones entre los mismos se encuentra optimizada para el problema en cuestión tal y como se vió en el capítulo 3). Mantener la distribución de los datos de modo tal que se preserve la localidad es una tarea

de especial importancia especialmente en sistemas donde la carga de trabajo (*workload*) debe ser continuamente reasignada.

Generalmente las estrategias de balanceo de carga se basan en la diferenciación entre procesadores ociosos frente a procesadores extremadamente sobrecargados de trabajo. Esta diferenciación es la que guía la migración de carga de unos procesadores a otros (de los cargados a los ociosos) en el momento en el que se toman las decisiones de equilibrado. La decisión o la tarea del balanceador consiste en decidir qué mapa de procesadores es el más idóneo para la computación en curso, tras la labor del balanceador viene la tarea de *mover la computación* entre los procesadores (siguiendo las instrucciones del balanceador) de modo que se alcance una situación de carga homogénea [201]. Por norma general estas estrategias no toman en consideración la relación que existe entre las hebras de cómputo (u objeto, en el caso de Charm++) y la decisión del balanceador se limita a aliviar los procesadores de carga pero no a decir qué hebra sería conveniente extraer, es decir, que las estrategias de balanceo generales no consideran la localidad de los datos como criterio complementario en la decisión de balanceo y según el tipo de aplicación, esto puede redundar en un incremento en las comunicaciones (si se separan hebras que han de comunicarse) y un decremento en el rendimiento.

El problema del balanceo de la carga ha sido abordado desde diferentes aproximaciones. Es posible abordar el balanceo usando librerías dinámicas y gestionadas por un runtime donde la inclusión de estrategias se realiza de manera sencilla o es posible también el extremo opuesto, donde se puede dotar a la propia aplicación de la utillería necesaria para decidir su mapa de computación [202]. En este trabajo hemos decidido optar por usar el runtime que se empleó durante el capítulo 3 para incluir nuestras estrategias de balanceo. Las razones que nos llevaron a esto fueron varias, entre ellas, que al ser un framework orientado a objetos la inclusión de nuevas estrategias de balanceo se hace sin tener que alterar las bases de datos de rendimiento que ya posee el runtime, para esto hacemos uso de la *herencia*. Por otro lado, el runtime adquiere perfiles que almacena en la base de datos y provee de medios que sirven para instrumentar a la aplicación en curso, por lo que no es necesario hacer que cada aplicación que deseemos ejecutar en el cluster deba tener implementadas rutinas de lectura de rendimiento, de contadores de la cpu, etc... todo esto ya lo provee el runtime. Una característica importante y muy relacionada con la labor de los balanceadores de carga, se encuentra en la definición del concepto de *carga de trabajo o workload* y en la del concepto *background load o carga de trabajo ajena a nosotros*. Una

extensa revisión de la literatura invita a que estos conceptos sean definidos en función del objetivo que cada aplicación persiga. Por esto proveer un medio flexible donde poder reimplementar conceptos es un punto a favor. En este caso el runtime AMPI favorece todo esto. Además este framework usa a las hebras como entidad computacional. Abordar el problema del balanceo de la carga usando un runtime o framework como el ofrecido por AMPI supone beneficios como la capacidad de variar, con relativa facilidad, el grano de la aplicación paralela; técnica con un impacto directo en la caché. Otra ventaja más que nos proporciona es un medio sencillo para materializar la migración de la carga entre procesadores, ya que la migración significaría un intercambio de hebras (stack, registros, etc...) entre procesadores. Las nuevas plataformas hardware como los procesadores multicore, dada su arquitectura basada en múltiples flujos de control, son entornos en los que este tipo de frameworks (como vimos en el capítulo 3) pueden explotar el rendimiento.

La complejidad en el uso de los clusters crece en la medida en la que los recursos de un cluster se reparten entre más usuarios. Esta situación es especialmente cierta en los clusters de estaciones de trabajo, donde la carga varía aleatoriamente dependiendo del número de tareas activas. Esta situación genera el desequilibrio extrínseco a la aplicación. Existe otro tipo de desequilibrio, denominado intrínseco que depende de los datos que el algoritmo maneje. Existe otro tipo de balanceo que podemos denominar intrínseco es especialmente notable cuando se trabaja con estructuras de datos como las matrices dispersas. En esta situación gestionar el desequilibrio es más sencillo que en el caso extrínseco donde se deben ajustar parámetros como la frecuencia de activación del balanceador, etc.... En el caso de la aplicación de reconstrucción 3D pueden aparecer ligeros desequilibrios intrínsecos pues es probable que haya elementos de procesamiento que converjan antes que el resto. Ante el desequilibrio extrínseco se debe tener en cuenta que cualquier política de balanceo empleada ha de respetar la localidad de los datos para así evitar un incremento en las comunicaciones que penalizará el rendimiento ante las altísimas latencias de la red.

Este capítulo presenta y analiza estrategias dinámicas de balanceo de carga que preservan la localidad de los datos. Además evalúa su aplicación a problemas modelados como SPMD (Single Program Multiple Data) y ejecutados en clusters. En este capítulo se aportan dos estrategias de balanceo de carga. La política de planificación de ambas se ha inspirado en dos algoritmos genéricos de balanceo de carga (Rake y Sliding) [203], estos algoritmos fueron diseñados

originalmente para resolver problemas de desequilibrio intrínseco en computadores SIMD. Los algoritmos han sido adaptados y mejorados para implementar el balanceo, en clusters no dedicados, de aplicaciones de tomografía. una de las mejoras incluidas en estos algoritmos consiste en estructuras de datos (componente esencial en estas estrategias) para forzar la localidad de los datos, implementando una ordenación FIFO de las tareas a ser migradas entre procesadores (virtualmente enlazados). Debido a que este enlace virtual entre procesadores es determinante en el movimiento de las hebras (objetos), las estrategias se denominan RakeLP y SlidingLP (donde LP hace referencia a Locality Preservation).

Llegados a este punto se puede crear la siguiente duda: *Si la estructura con la que trabajan los balanceadores es FIFO y se instaura el orden dentro de esta estructura enlazando a procesadores virtualmente, entonces ¿no se contempla migrar una hebra en función de su intensidad de cómputo sino de su posición en la estructura? Y si esto es así, entonces ¿El desequilibrio intrínseco no se contempla?* Lo cierto es que nuestra computación es regular para cada hebra (no obstante para estudiar este efecto se ha implementado un monitor gráfico –***PerfVis***–). Por esta razón si se da el caso de que una hebra (u objeto embebido en una hebra) computa menos que el resto (por desequilibrio intrínseco) entonces ese procesador físico registrará más capacidad para albergar hebras y lo que conseguiremos es enviar al procesador físico donde está el objeto que ya ha terminado, más hebras de cómputo. Desde esta perspectiva, el desequilibrio intrínseco está controlado. Por otro lado, si se genera una situación complicada (en cuanto a computación) dentro de un procesador físico por computación de otros usuarios, entonces se extraerán entidades de procesamiento y se enviarán al siguiente procesador enlazado (siempre respetando el orden FIFO que respeta la localidad de los datos). En este caso también se prevén los desequilibrios extrínsecos.

Para implementar estas estrategias de balanceo sensibles a la localidad de los datos, se ha seguido un esquema centralizado, esto es, la decisión de balanceo se toma en un único procesador que tiene acceso al perfil de rendimiento de todos los involucrados en la computación. Para conseguir esto ha bastado crear interfaces de acceso a las tablas de rendimiento que mantiene el runtime usado (AMPI–Charm). Una de sus principales ventajas es que, como vimos en el capítulo 3, que permite portar aplicaciones MPI al entorno multihebrado donde podremos habilitar políticas de balanceo de carga. Usar el runtime AMPI–CHARM además nos ha provisto de la utilisima abstracción que empleamos en el capítulo 3 de esta tesis. Al disponer de esta abstra-

cción se ha podido embutir a cada proceso MPI en una hebra y así considerar a cada procesador físico usado como un procesador multihebrado. La agilidad en el empleo de las hebras vista en el capítulo 3 (cambios de contexto para agilizar la ocultación de latencias y planificadores ad–hoc para escoger en cada caso la política adecuada) se amplía con la capacidad de poder *mover* hebras de usuario (carga computacional) entre procesadores con mayor agilidad que si se tuviera que mover procesos en aplicaciones SPMD [204] [205].

La librería de paso de mensajes MPI tiene un gran desventaja y es que no considera implícitamente la concurrencia en un proceso. En general cualquier aplicación MPI posee una única hebra de control por procesador, es decir, un único proceso en ejecución por nodo. Uno de los principales inconvenientes es que un proceso completo puede ser bloqueado cuando se efectúe una operación de entrada–salida. Aspecto que como se ha demostrado puede ser obviado en entornos multihebrados. Otra ventaja que ofrecen las hebras de usuario es que pueden ser monitorizadas con bastante facilidad. Y junto con las políticas de balanceo (desarrolladas a voluntad) se puede llegar a manejar o explotar con acierto la concurrencia. Como nota, recordaremos que AMPI es la capa (o framework) de más alto nivel dentro de las que forman el RTS Charm++ cuyo núcleo es un runtime denominado Charm. Una de las principales características de la capa AMPI es que adapta procesos MPI (especialmente los desarrollados bajo el estándar MPI–1) para que sean embebidos en hebras de usuario. Como ya se ha demostrado con anterioridad, las hebras de usuario son superiores a los procesos en aspectos clave para nosotros como el cambio de contexto y la migración.

Este capítulo se organiza como sigue, primero se plantea en la sección 4.2 la necesidad de que las aplicaciones científicas puedan equilibrar su computación de acuerdo a sus necesidades. En la sección 4.3 se plantea la forma en la que podemos incluir estrategias de balanceo en las aplicaciones científicas sin alterar su lógica. En la sección 4.4, se describen las estrategias de balanceo de carga diseñadas para BICAV. Posteriormente, en la sección 4.5 se evalúan y analizan nuestras implementaciones en términos de granularidad y número de procesadores físicos. Las evaluaciones de rendimiento han sido llevadas a cabo empleando primero aplicaciones test (algoritmo de relajación basada en Jacobianos 3D [206], ver capítulo 1, que consiste en un método iterativo clásico para resolver ecuaciones diferenciales parciales que muestran fuertes propiedades de localidad en los datos) cuyos patrones son idénticos a nuestra aplicación de reconstrucción (BICAV portada al entorno multihebrado). Tras estas evaluaciones donde se puede comprobar

la validez de las estrategias, se presenta un estudio donde se analiza el comportamiento de la aplicación de reconstrucción ante situaciones de desequilibrio, resueltas con nuestras estrategias. Finalmente se recoge en la sección 4.8 de conclusiones adquiridas tras el trabajo con los balanceadores en aplicaciones de reconstrucción de imágenes.

4.2. Balanceo de carga en Aplicaciones científicas

Escribir una aplicación paralela significa añadir una capa extra de complejidad al desarrollo del software. Esta complejidad hace referencia a determinar, no sólo cuando será ejecutada una operación determinada sino dónde -procesador- será llevada a cabo. Si hay algo que caracteriza a una aplicación paralelizable es que a menudo su marcado carácter irregular hace aún más difícil una paralelización eficiente. Es en este aspecto donde la decisión de quien la paralelice puede afectar en el resultado final. Se puede optar por realizar una distribución de la computación sin tener en cuenta el posible desequilibrio, generando un programas con problemas de rendimiento. Por otro lado también se puede optar por incluir, en el código de la aplicación misma código de balanceo de carga, incrementando innecesariamente el tiempo de desarrollo. Esta última opción es la más acertada pero no es muy aconsejable que la aplicación y el balanceador se fusionen en un único código. Las razones para aconsejar esto van desde la dificultad de mantenimiento de la aplicación (ya que la lógica se ha visto ampliada y modificada para albergar al balanceador), por otro lado hacer cambios en la estrategia de balanceo supondría realizar modificaciones en el código fuente de la aplicación, lo que unido a la dificultad de mantenimiento puede resultar en modificaciones con resultados inesperados. En este sentido incluir la posibilidad de que el código de una aplicación invoque a una estrategia de balanceo, cuyo código no se mezcle con el de la aplicación es además de recomendable una verdadera ventaja. El problema que aparece entonces es el de la inclusión de nuevas estrategias de balanceo. Usar la Orientación a Objetos nos permite hacer uso de la característica de herencia para poder incluir balanceadores de carga sin más que hacerlos depender de la jerarquía ya existente.

El siguiente paso a discutir cuando se aborda el desequilibrio de una aplicación paralela pasa por cómo afrontar la migración de computación. La capacidad de mover carga computacional entre procesadores es prácticamente un esquema consensuado ([207], [208], [209], [210]) para afrontar las situaciones de desequilibrio ya sean generadas por la misma aplicación o por sucesos externos a la misma (diferentes frecuencias de reloj, procesadores usados por más de una

aplicación, etc).

Ejecutar la aplicación en procesadores con arquitecturas diferentes, por ejemplo, *PowerPC* y *Cell* simultáneamente puede ser complejo, especialmente desde el punto de vista de la migración. Pues a pesar de que el runtime subyacente actúa como interfaz con la arquitectura real, no la *abstrae* como intentamos realizar nosotros con Java (ver Capítulo 3). Por tanto trasladar la ejecución de un procesador a otro puede llegar a ocasionar inconsistencias. Si el contador de programa se detiene en una instrucción concreta en el procesador A, ¿a qué instrucción del procesador B he de ir si esa instrucción de A equivale a dos en B?

Un balanceador tiene la misión de encontrar la mejor asociación existente entre procesadores y componentes de nuestra aplicación. La estrategia de balanceo que diseñemos ha de estar íntimamente ligada al tipo de aplicación que queramos balancear. Existen dos esquemas clásicos de balanceo de carga [1], el esquema estático donde no se necesita cargar al sistema con la inclusión de un *runtime* que evalúe continuamente el patrón de ejecución de la aplicación y donde el reparto de carga se hace distribuyendo bucles de la computación cíclicamente o por bloques si la arquitectura es UMA o siguiendo las restricciones impuestas por la distribución de datos si la arquitectura es NUMA; y el esquema dinámico útil cuando la duración de las iteraciones es desconocida a priori, en este caso se necesitan métricas de rendimiento. El balanceo o planificación dinámica se apoya en tres modelos: *predicción del futuro* que consiste en distribuir la computación de acuerdo con la potencia de cómputo de cada procesador y evaluar con una cierta frecuencia el rendimiento. En esta tesis se ha empleado este modelo [211]. *Cola de tareas* que se basa en mantener colas de trabajo capaces de asignar tareas a cada procesador según termina, modelo muy útil para los procesadores de memoria compartida. *Difusión* que consiste en distribuir la carga uniformemente y permitir el movimiento de la misma entre procesadores adyacentes hasta equilibrar en cada instante la carga del sistema.

El balanceo dinámico consiste en asignar a cada procesador una carga proporcional a la capacidad computacional del mismo, acortando así el tiempo de ejecución de cada iteración. En el balanceo dinámico se deben afrontar aspectos como la *sincronización* que consiste en detener la computación en una barrera para poder evaluar el rendimiento, por esto es importante buscar una solución de compromiso entre la frecuencia de la evaluación y la efectividad del balanceo. En este capítulo se presenta una posible solución a este problema. *Las métricas de rendimiento*, constituyen la clave para la efectividad del balanceo, en la medida en que tratamos de predecir

el futuro apoyándonos en la información de carga que recabamos del pasado. Las métricas que hemos empleado durante este trabajo se han basado en el tiempo que permanece ocioso cada procesador, en el tiempo de CPU consumido por cada unidad computacional y las iteraciones por segundo (o fracción de segundo por iteración). *La migración de carga y análisis de viabilidad*, la migración es el medio con el que hacer efectivo el balanceo de la carga. Mover la carga implica también mover los arrays de datos que nutren a cada iteración. Evidentemente existe una relación de compromiso entre el balanceo y la migración de carga. Analizar el coste y el beneficio de esta operación es un trabajo sutil, existe un análisis extenso sobre qué parámetros estudiar en [1]. Las razones son que las imprecisiones que se cometan al calcular los costes asociados al movimiento de la carga pueden cancelar los beneficios potenciales de la migración. En nuestro esquema, la migración. Una práctica muy extendida es realizar la migración si la mejora supera en un 10 % la situación actual [1].

En el balanceo dinámico es muy importante la estrategia que se emplea pues es quien ha de resolver el problema de la reasignación. Las estrategias pueden ser *globales* o *locales* dependiendo de la información que se maneja en cada momento. Y pueden ser centralizadas o distribuidas, en función de cómo se organice la estrategia, es decir, si se encuentra en un procesador (centralizada) o si se encuentra repartida entre varios procesadores. La tabla 4.2 presenta una breve descripción de cada tipo de estrategia que se puede seguir en el balanceo dinámico.

La ventaja de las estrategias globales es que la distribución del trabajo es óptima (basándonos en la información útil hasta ese momento). Sin embargo, la sincronización es más costosa (ver figura 4.2). En los esquemas locales, la convergencia (distribución de la carga) es más lenta. Sin embargo, el coste de la sincronización es menor. Un defecto de las estrategias locales es que se puede producir desbalanceo entre grupos de procesadores [1]. A pesar de que existen muchos tipos de aplicaciones diferentes, y cada una sigue un patrón diferente, si es cierto que las aplicaciones científicas tienen algunas características en común, como son:

- La computación realizada por cada aplicación es iterativa, llegando a repetir la misma computación decenas de miles de veces.
- La duración de la iteración puede variar ampliamente, por ejemplo de 1 segundo a 1 minuto en procesos independientes.
- Las iteraciones no pueden solaparse significativamente, además las dependencias en las

Estrategias dinámicas	Global	Local
Distribuida	El perfil de información se envía a todos los procesadores (broadcast). No hay necesidad de esperar instrucciones de un procesador	Se forman grupos de k procesadores siguiendo algún criterio (proximidad física, p.e.). El balanceador se replica en todos los procesadores pero la información de rendimiento sólo se comparte entre procesadores del mismo grupo.
Centralizada	El balanceador se encuentra en un procesador *maestro*. Cuando se activa la estrategia, los demás procesadores quedan a expensas de las órdenes de este maestro.	El procesador más rápido de cada grupo interrumpe al resto y se encarga de redistribuir la carga.

Tabla 4.1: Descripción de estrategias

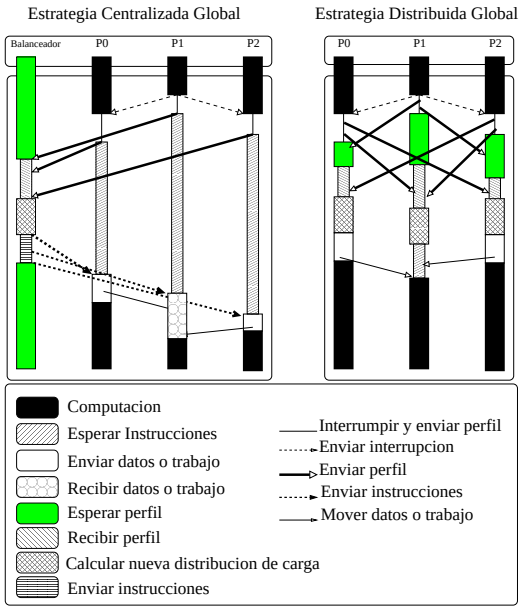


Figura 4.1: Comparación entre las estrategias globales centralizadas y ditribuidas [1].

comunicaciones hacen que los procesadores estén prácticamente sincronizados.

- La carga de trabajo de la aplicación (e incluso la que otros usuarios pudieran inducir) no cambia drásticamente.

- La E/S no es un factor determinante en el rendimiento de las aplicaciones. Bien porque apenas hay o bien porque el efecto adverso que generan puede ocultarse.

Un amplio porcentaje de las aplicaciones paralelas actuales apenas son síncronas, es decir, la computación llevada a cabo en un procesador depende estrechamente de los datos que se generan en otros procesadores. Considérese el caso de nuestra aplicación de tomografía, BICAV, donde en cada procesador se trabaja con un *slab* –conjunto de *slices* o porciones del volumen–, para poder definir la relación entre cada rayo de proyección y cada *slice* es preciso discretizar el volumen. La elección de esta figura de discretización, como se vió en el Capítulo 2, afecta a la calidad de la reconstrucción. Si se escoge el voxel, entonces cada procesador es independiente de los demás en todo el proceso de reconstrucción, pero la calidad es baja. Si se escoge el blob, entonces se establecen etapas de comunicación entre procesadores durante cada iteración de la reconstrucción, ya que el blob puede tener un radio superior al de un *slice* y por tanto afectar a *slices* de otros procesadores. El grafo de dependencia indica que ningún procesador puede progresar ni un sólo paso más allá que el procesador más lento (ver capítulo 3). Incluso hay momentos en los que se aplican reducciones globales donde los procesadores experimentan una dependencia mayor. En este caso, en el de las computaciones síncronas globales, usar un cluster donde los procesadores son compartidos, e incluso un multicore donde se puedan lanzar varios trabajos a la vez hace que se presenten problemas de desbalanceo. Considérese por ejemplo, que se lanza la aplicación en ocho procesadores del cluster y que se dispone de la totalidad de la eficiencia de los procesadores. Es probable que durante la ejecución del trabajo, otro usuario lance otro trabajo y que esto hace que en un procesador sólo dispongamos de la mitad de eficiencia mientras que en el resto de procesadores se cuenta con toda la potencia inicial. No obstante y debido a las dependencias establecidas en la computación el resto de procesadores deberán emplear la mitad de su tiempo útil en esperar a los resultados del procesador más lento. Aunque parezca que se ha perdido cerca de 1/16 del rendimiento total de la aplicación, realmente el rendimiento del programa se está viendo afectado en un 50 %.

Para estudiar cómo se comportará nuestra aplicación BICAV ante situaciones de desequilibrio

y predecir las ganancias al implantar una determinada estrategia de homogeneización de la carga, hemos usado como *benchmark* particular una implementación de Jacobi en tres dimensiones (ver capítulo 1). Este benchmark tiene la característica de comportarse de manera idéntica, en cuanto al patrón de comunicaciones, a nuestra aplicación. La dependencia entre los datos es más compleja, pues implementa una malla 3D cuando en el caso de BICAV la dependencia se puede implementar linealmente. En este *benchmark* el dominio computacional es una malla tridimensional que se divide en celdas. Las celdas se distribuyen entre los procesadores. Esta malla se usa para *propagar* ciertos valores inducidos en varias regiones de la malla. El programa sigue un modelo iterativo, en el que en cada iteración se emplea para actualizar los valores de todas las celdas de la malla. La difusión (o actualización) de estos valores sigue un esquema basado en la relajación Jacobiana, cada celda actualiza su valor en base a la media de los valores de las celdas vecinas. El esquema de relajación se repite hasta que se alcanzan los criterios de convergencia establecidos. Para evaluar la convergencia (igual que en BICAV) se deben emplear reducciones globales para determinar el error global, situación en la que la iteración termina y se procede hacia la siguiente iteración.

Tras examinar la naturaleza de las aplicaciones científicas, las plataformas sobre las que estas se ejecutan y la forma de proveer a las mismas de un medio para adaptarse al medio sin lastrar la eficiencia, se presentará una breve descripción del framework usado.

4.2.1. La migración y la planificación de procesos como solución a las necesidades de adaptabilidad

A pesar de que el problema de *balanceo de carga* y el de *compartición de carga* son dos problemas diferentes, es cierto que del segundo se pueden extraer muchas conclusiones útiles para el primero. La compartición de carga se plantea como solución al problema de extraer el máximo de la potencia de procesamiento de un sistema distribuido. Para distribuir o repartir la carga (que no balancearla) se deben plantear estrategias que definan qué es carga y cómo conseguir medirla, obviamente el concepto de carga en un sistema depende de la definición que se le de, por ejemplo, existen grupos de investigadores que deciden instrumentar las instalaciones de los sistemas operativos Unix/Linux de las máquinas para extraer de ahí indicios de utilización y rendimiento; por ejemplo, Eager et al en [212] realizaron simulaciones de diferentes estrategias de distribución de carga. Para Zhou, en [213], la carga del sistema se podía identificar con la

longitud de las colas que cada recurso del sistema operativo registra. Y fue precisamente Zhou quien más tarde propuso varios algoritmos [214] con los que poder compartir la carga entre los nodos de un cluster (fue él quien presentó dos diferentes grupos de estrategias de balanceo, *estrategias centralizadas* y *estrategias distribuidas basadas en vecinos*). Leland y Ott en [215] también estudiaron el comportamiento simulado de estrategias de compartición de carga sencillas, pero en este caso extraían la información de rendimiento de trazas de rendimiento instaladas en los mismos sistemas operativos que empleaban. Aunque existen diferencias significativas entre el problema de repartir la carga y el problema de balancear la carga en aplicaciones paralelas (en estas aplicaciones -aplicaciones paralelas- ***la dependencia de los datos y*** **la localidad** son aspectos mucho más importantes que el estudio del uso de los recursos del *sistema operativo* a la hora de determinar el rendimiento de la aplicación en un procesador.

Por otro lado, en los trabajos [216], [217], sus autores (Devarakonda e Iyer) implementaron mecanismos de medida en sistemas Unix para demostrar que el uso futuro de los recursos puede predecirse en base a las medidas de rendimiento que se obtuvieron en instantes anteriores. Es decir, con las medidas de rendimiento se pueden crear estimaciones sobre el futuro de la computación (recuérdese que en este sentido los objetos son altamente predecibles). Apoyándose en esta aproximación Goswami et al. [218], [219] describieron un conjunto inicial de algoritmos de compartición de carga que basan la decisión de dónde iniciar un proceso en base al estudio del uso de recursos de un procesador. Además consiguieron demostrar con simulaciones basadas en trazas que se obtenían mejores rendimientos considerando este histórico que sin considerarlo. Baxter y Patel [220] presentan un algoritmo para distribuir grafos paralelos y dinámicos de tareas. El uso de estos grafos está muy extendido para ciertos tipos de aplicaciones paralelas. Estos grafos de tareas se distribuyen estáticamente, tras hacer esto se obtienen trazas de rendimiento a través de simulaciones. Estas trazas se usan para guiar a las estrategias de migración de carga en los algoritmos de balanceo de carga, que usando exclusivamente data de medidas en anteriores ejecuciones e información de cada procesador (entorno de ejecución local) arroja rendimiento simulado cercano al obtenido usando un algoritmo ideal dotado de información global. La validez de la predicción del comportamiento futuro de procesos Unix y la eficacia de las técnicas de distribución (o compartición) de carga usando predicción y medidas de rendimiento es una evidencia firme de que una técnica similar pudiera ser útil para programas paralelos basados en objetos.

La planificación de procesos ha sido un tema ampliamente investigado, especialmente por que cada vez más se ha dado en usar máquinas paralelas de bajo coste y mantenimiento, caracterizadas por disponer de enlaces de comunicaciones muy lentos, de modo que compartir los procesadores existentes entre varias aplicaciones ha hecho necesario desarrollar teorías sólidas en este campo. Corregir la planificación de procesos inicial implica tener que mover procesos entre procesadores. Las citadas latencias de red, precisamente, han hecho que como consecuencia este punto también haya sido extensamente estudiado.

La migración de procesos ha sido una solución muy recurrida para solventar problemas de balanceo de carga y para dotar de *tolerancia ante fallos* a aplicaciones paralelas (o para un conjunto no relacionado de procesos). La implementación tradicional de esta técnica en clusters homogéneos ha sido la de portar completamente un proceso de procesador a procesador (salvando dificultades como las presentadas por los manejadores de archivos abiertos por cada proceso, como ejemplo se puede tomar el trabajo de Sprite [221] o la implementación de MOSIX [222], Condor [223] o Piranha [224]). La imagen de un proceso es bastante *pesada* y por tanto la migración siguiendo esta tónica es costosa en tiempo. A parte de los costes de tiempos asociados a las latencias de red y del tiempo que el *cedular* del sistema operativo de destino tarde en planificar de nuevo al proceso recién migrado.

Evitar el uso de procesos tal y como se entiende en un SO como Linux o Unix es primordial para poder ganar rendimiento en aplicaciones desbalanceadas sin tener que pagar un extra por cada migración que necesitemos realizar para alcanzar la situación de balanceo óptima. A este efecto, existen numerosas implementaciones de frameworks o librerías que sustituyen el concepto de proceso por el de proceso ligero (o hebra, u objeto) según convenga. Algunos de los más conocidos son UPVM [225] -que emplea hebras para migrar carga y un modelo de programación similar a PVM-, Charm [226] -que es parte del framework que usamos, combina las dos opciones: uso de objetos, o de hebras que abstraen procesos MPI,además implementa un útil modelo de computación asíncrono basado en paso de mensajes-, ELMO [227] -librería que usa Charm para construir funciones de tipo SPMD, como recepciones bloqueantes-, Cilk [228], que es framework de programación paralela y multihebrada, no se apoya en el concepto de objetos aunque explota las abstracciones basadas en hebras, en Cilk la computación se divide en hebras que se distribuyen entre los procesadores siguiendo un esquema de *work-stealing*, es decir, el procesador que agota su computación pide más hebras al sistema. Cilk es bastante potente

pero no nos permite la flexibilidad que nos da Charm. Desde el lado exclusivo de la orientación a objetos, también existen implementaciones que ofrecen objetos paralelos y distribuidos como lo hacen POOMA [229], o DOME [230], pero de nuevo, no nos ofrecen la posibilidad de adaptar nuestras aplicaciones implementadas en MPI.

¿Es factible ignorar a los frameworks y confiar en los compiladores?

Existen compiladores que tienen extensiones para paralelizar, y en esta paralelización se pueden incluir algunas directivas para el balanceo de la carga, pero la mayoría de los compiladores emplean información disponible en tiempo de compilación. En arquitecturas SMPs los compiladores con extensiones para la paralelización pueden hacer uso de paquetes de *hebras de kernel* con el fín de enviar carga a un conjunto de hebras y de ir manteniéndolas ocupadas con más trabajo cuando vayan quedando liberadas de trabajo. No obstante estas hebras ni son susceptibles de ser migradas ni son tan flexibles como las hebras de usuario.

4.3. Framework de balanceo como soporte a la adaptabilidad

Aunque la carga de una aplicación puede verse afectada por diversas razones, es posible llegar a analizar la carga a través de medidas de tiempo (consumo de CPU, tiempo que la aplicación está ociosa, etc). Por otro lado, considerar que la aplicación no es ni más ni menos que un conjunto de objetos agiliza y facilita la separación entre el proceso de decisión (balanceo de carga) del mecanismo que lo materializa: migración de carga.

Evidentemente hay que definir qué es carga para una aplicación y qué es carga de una aplicación. La definición de carga ha de ser lo más independiente posible de la aplicación. Una opción plausible es la de emplear el tiempo de ejecución de cada objeto en que se divide la computación. Además se debe considerar el patrón de comunicaciones entre objetos pues también afecta al rendimiento. Es posible considerar que el grano de los objetos es lo suficientemente fino como para entender que migrando objetos es posible controlar el efecto de la carga computacional.

Para balancear aplicaciones contamos con tres componentes elementales: *la base de datos de balanceo, la estrategia de carga y el sistema de coordinación de carga.* El coordinador de carga es parte del framework y su misión consiste en proveer la interfaz entre aplicación y estrategias de balanceo. El coordinador de carga emplea a la base de datos de carga para realizar el seguimiento de cada objeto y de cómo y con quien se comunica, además de ser parte importante en la

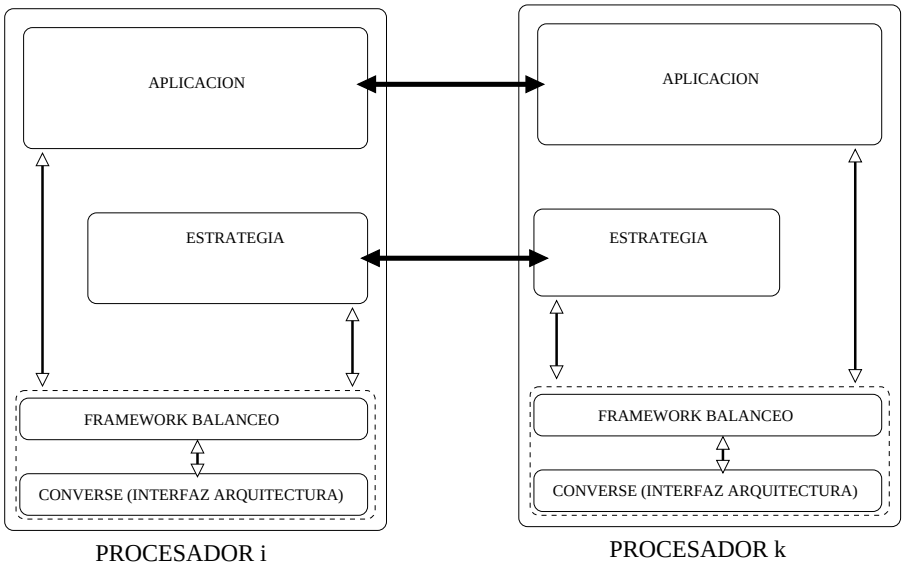


Figura 4.2: Interacción balanceador y aplicación

etapa de migración. La base de datos de carga, se encarga de mantener estructuras de datos (figura 4.2) que son mantenidas por Converse (figura 4.3). La estrategia de balanceo de carga es el componente del *framework* que se encarga de *analizar* la información almacenada (por el coordinador de carga) y toma las decisiones sobre las migraciones y qué objetos habrán de migrar.

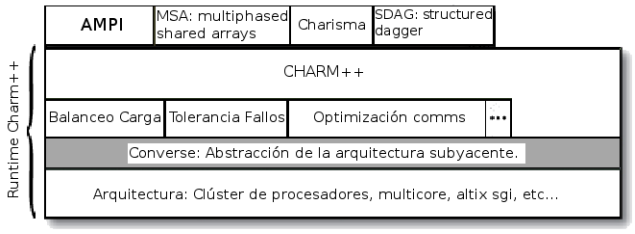


Figura 4.3: Capa Converse del Runtime Charm++.

En la figura 4.4, se puede comprobar cómo se relaciona la computación con el framework de balanceo. En la capa superior se encuentra la *estrategia de balanceo de carga*. El *coordinador de carga* indica a la estrategia sobre cuando es el mejor momento para iniciar un estudio del balanceo de la carga. Tras cada notificación del *coordinador* cada procesador detiene su computación para dejar que la estrategia recopile la información de rendimiento que se almacena en la *base de datos* (como por ejemplo el *tiempo total de Walltime* y *el tiempo de CPU* consumidos desde el anterior balanceo, cuanto tiempo estuvieron *ociosos*, y la reestimación de la velocidad del

procesador). La estrategia también dispone de información sobre las comunicaciones (parciales a cada procesador) que mantienen los objetos (con quien se comunican los objetos, cuanta información envían, etc). La estrategia se encarga de decidir qué objetos han de cambiar de localización. El coordinador de carga se encarga de monitorizar las migraciones. La forma en la que este *coordinador* actúa con los demás objetos es a través de los *gestores de objetos*, de modo que cuando el coordinador de carga recibe (de la estrategia) la orden de migrar un objeto, éste se dirige a los gestores de objetos, que son quienes coordinan la migración con el objeto del usuario. Y al revés, es el *gestor de objetos* quien solicita al *coordinador de carga* la intervención de la estrategia, cada cierto número de iteraciones o en determinados puntos de sincronización.

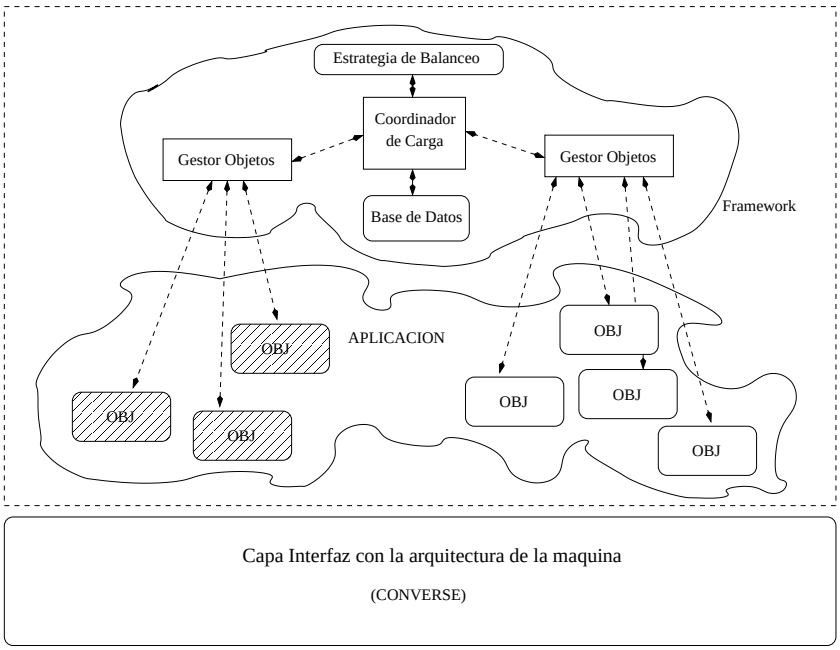


Figura 4.4: Framework balanceador y aplicación

Es preciso considerar que como el código de la aplicación es parte de cada objeto, se debe evitar poner llamadas explícitas al coordinador de carga directamente desde este código. Es el *gestor de objetos* quien se encarga de esto. La interfaz entre objetos de la aplicación y gestores de objetos depende del tipo de objeto usado. Por ejemplo, para nuestra implementación basada en Charm++ se debe proporcionar en la implementación de cada objeto dos métodos, el método *pack* y el método *unpack*. Métodos que el gestor invocará para realizar la migración. Esta funcionalidad, por ejemplo, es innecesaria si se emplean *hebras migrables* en lugar de objetos, como es el caso de AMPI, ya que en este caso se empaqueta todo el stack (salvo punteros a estructuras

en el heap). En la migración, es necesario diferenciar dos tipos de objetos : los objetos desatendidos, que pueden ser migrados en cualquier momento y los objetos *síncronos*, que precisan ser migrados exclusivamente en ciertos instantes de la computación, como es el caso que nos ocupa con BICAV (tanto en la implementación hebrada como en la implementación orientada a objetos). Los objetos *sincronizados* deben, por tanto, informar a su gestor –*en intervalos regulares*– sobre los puntos en los que estos objetos pueden ser migrados. Para esto deben usar una llamada a función especial que actúa como una *barrera local*. Estos objetos no pueden abandonar el *estado de migración* hasta que el *coordinador de carga* así lo indique. Una vez que todos los objetos sincronizados alcancen la barrera se podrá iniciar la actividad de balanceo ya que la estrategia se ejecuta cuando toda la computación se encuentra en la barrera establecida para el balanceo. Y por supuesto, si la estrategia determina que se deben llevar a cabo migraciones, se lo informa al coordinador de carga.

En nuestras aplicaciones encontramos varios parámetros a configurar. Por un lado se debe especificar la frecuencia con la que (ya sean hebras migrables u objetos) se ha de invocar al balanceador. Indicar un número de iteraciones puede afectar directamente al rendimiento de la aplicación, pues si el número que se especifica es elevado entonces estaremos interrumpiendo a la aplicación en aras de un hipotético desbalanceo. Por otro lado, si la frecuencia es demasiado baja podemos estar usando el balanceo menos veces de lo que la aplicación pudiera requerir, haciéndolo ineficiente. Por otro lado, el hecho de que exista una barrera de sincronización (invocada un número incorrecto de veces) hace que la computación se ajuste a la computación del procesador más lento. Como solución a este handicap se ha desarrollado una aplicación de monitorización que no interrumpe a la aplicación de usuario y que permite invocar al balanceador de manera flexible.

4.3.1. Consideraciones sobre la caracterización de los procesadores

Las aplicaciones que se ejecuten en máquinas paralelas no dedicadas han de considerar la carga de background para maximizar el rendimiento. Una solución interesante para *detectar* el background es identificar la situación en la que el procesador se comparte con otro proceso *computacionalmente intensivo* e iniciar la estrategia para considerar retirar nuestras hebras (u objetos) de ese procesador. Es obvio que cuando remita el *background* en ese procesador se debería contemplar volver a incluirlo en el mapa de procesadores hábiles para la computación

(aspecto que se tiene en cuenta en las estrategias que hemos desarrollado).

Nuestras ejecuciones y nuestro entorno de trabajo consiste en máquinas compartidas de propósito general, ya se trate de *clusters* o de *procesadores multicore* es por esto que hay que considerar ciertos aspectos, sobre todo en los clusters. Aspectos como que los procesadores posean arquitecturas compatibles pero con frecuencias de reloj diferentes. En este sentido el *framework* usado ofrece un factor de corrección que se obtiene tras evaluar con un *benchmark* inicial la potencia de cálculo de cada procesador.

El siguiente código se encarga de determinar la *capacidad de cómputo estimada* para cada procesador :

```
extern "C" int LDProcessorSpeed()
{
  static int thisProcessorSpeed = -1;
  if (_lb_args.samePeSpeed() || CkNumPes() == 1) return 1;

  if (thisProcessorSpeed != -1) return thisProcessorSpeed;

  static int result=0;

  int wps = 0;
  const double elapse = 0.2;

  const double end_time = CmiCpuTimer()+elapse;
  wps = 0;
  while(CmiCpuTimer() < end_time) {
    work(1000,&result);
    wps+=1000;
  }


  for(int i=0; i < 2; i++) {
    const double start_time = CmiCpuTimer();
```

```
    work(wps,&result);
    const double end_time = CmiCpuTimer();
    const double correction = elapse / (end_time-start_time);
    wps = (int)((double)wps * correction + 0.5);
  }

  thisProcessorSpeed = wps;

  return wps;
}
```

Este código lo ejecuta cada procesador, si es un SMP entonces sólo lo ejecuta uno de los procesadores y la medida se hace extensible para todos. Se pretende medir cuantas iteraciones se registran en 0.4 segundos. Hay muchas llamadas a función por lo que resulta en bastante trabajo. Habiendo ejecutado el primer bucle ya se posee una estimación para las iteraciones posibles en un segundo. Durante el segundo bucle se efectúa trabajo durante algunos ciclos más de reloj, esto se hace para completar hasta *un segundo*, y así poder estimar correctamente el número de iteraciones por segundo.

La función a la que se llama para comprobar la velocidad del procesador es

```
static void work(int iter_block, int* result) {
  int i;
  *result = 1;
  for(i=0; i < iter_block; i++) {
    double b=0.1 + 0.1 * *result;
    *result=(int)(sqrt(1+cos(b * 1.57)));
  }
}
```

4.3.2. Estrategias de balanceo

El framework de balanceo incluye estrategias centralizadas tanto locales como globales. Las estrategias implementadas en el sistema incorporan como método de sincronización una barrera

global. Todas las estrategias, tanto las desarrolladas como las que se desarrollen, son orientadas a objetos y heredan de la clase CentralLB, esta clase implementa la coordinación y la comunicación para las estrategias de balanceo donde cada estrategia debe implementar obligatoriamente el método *work*, durante la ejecución de éste método se recopila información de rendimiento y se decide si habrá o no reorganización del trabajo. CentralLB recibe la señal de *sincronización* procedente de cada procesador que ejecuta la aplicación. CentralLB recoge el perfil de cada procesador y lo remite al procesador indicado como procesador que ejecutará la estrategia. Cuando este procesador recibe los perfiles, entonces iniciará el método *work* que se define para cada estrategia de balanceo que se desee implementar. En éste método se evalúa la información y se generan decisiones de migración. Estas decisiones son enviadas a CentralLB desde el procesador que evaluó la situación. Cada procesador actuará en consecuencia y consonancia a las órdenes dictadas por el procesador maestro.

El framework cuenta con las estrategias más conocidas ya implementadas. Estas estrategias son (destacamos las más interesantes para nuestro trabajo):

- *Random* Realiza una redistribución de la carga de manera aleatoria. Ignora por completo la información de los perfiles de rendimiento de cada procesador. Es un algoritmo de balanceo de complejidad O(N) ya que sólo requiere una única pasada por la lista de objetos. El balanceo puede ser razonable cuando las comunicaciones y su penalización son nimias y cuando hay un gran número de objetos.

- *Greedy* Es la implementación de una estrategia voraz sobre la clase CentralLB. No se considera el grafo de comunicaciones entre objetos. En su lugar se construyen dos listas, una de (concretamente un heap) procesadores donde el procesador con menos carga se encuentra en la cima. No hay asociación alguna entre procesadores y objetos. La segunda lista es otro montículo, en este caso, de objetos encontrándose en la cima de este montículo el objeto que más recursos haya consumido. En un bucle se va asignando el objeto más computacionalmente pesado con el procesador que menos carga ha tenido. Se reajusta la carga y se continúa hasta terminar con una asignación equilibrada. Si en la computación participan N objetos, entonces la complejidad del algoritmo es O(N log N) y además casi todos los objetos han de ser migrados por lo que el tiempo empleado por esta estrategia se ve incrementado.

Algoritmo 4.1 : Estrategia Refine.

```
1.  umbral = overload· averageload;
2.  P_sobrecargado ← procesadoresConCarga > umbral;          (montículo max-heap)
3.  P_infracargado ← procesadoresConCarga < umbral;          (montículo min-heap)
4.  while (P_sobrecargado no vacio)
5.    P_s ← procesador mas cargado en P_sobrecargado;
6.    Objetos_s ← objetos almacenados en P_s;                (montículo max-heap)
7.    repeat
8.      if Objetos_s esta vacio
9.        if carga(P_s) > umbral                    el procesador está sobrecargado
10.         return error                          y no podemos eliminar más carga
11.       endif
12.     else
13.       Objeto ← objeto mas pesado de Objetos_s
14.     endif
15.     P_i ← procesador más ocioso de P_infracargado
16.     if (carga(P_i) + carga(objeto)) < umbral
17.       eliminar objeto de P_s y recalcular la carga para P_s
18.       insertar objeto en P_i y recalcular la carga para P_i
19.     endif
20.     Eliminar objeto de Objetos_s
21.   until carga(P_s) < umbral
22.   Eliminar P_s de P_sobrecargados
23. endwhile
```

- *Refine* Esta es una estrategia que mejora el balanceo de la carga, refinando la distribución de objetos incrementalmente. Estas estrategias se invocan con un parámetro de tolerancia que se encuentra entre el 1.05 y 1.10 (de acuerdo a [211]). Este factor de tolerancia indica la proporción en la que un procesador determinado puede exceder el valor de la carga media global del sistema sin tener que replantear una reordenación de los objetos que en él se están ejecutando. En el Algoritmo 4.1, se muestra cómo trabaja esta estrategia. La efectividad de esta estrategia reside en que sólo los procesadores sobrecargados son sometidos a examen.

4.4. Balanceo de Carga y Localidad de los Datos

El objetivo de este trabajo ha sido generar algoritmos o estrategias de balanceo de carga útiles para entornos multiusuario en plataformas cluster. De este modo cualquier aplicación cuya distribución de datos se asemeje a una cola FIFO pudiera mantener siempre el mejor *mapeo* de

procesadores posible dadas las circunstancias de carga. En este sentido, las implementaciones de los algoritmos *Rake* y *Sliding* (ver [203]) han sido modificadas para considerar aspectos propios de los clusters no dedicados y además han sido adaptados para facilitar su inclusión en el framework AMPI–Charm. RakeLP y SlidingLP son implementaciones de estos algoritmos.

La evaluación de RakeLP y SlidingLP se ha llevado a cabo en un entorno multi usuario, la instrumentación de la computación y el soporte para tomar las decisiones de balanceo se ha hecho a través de los servicios que proporciona la capa de más bajo nivel de este framework, que se denomina Converse [231]. Converse es la capa que realmente hace las veces de intermediaria entre la máquina y el resto del runtime, es por tanto la capa que proporciona servicios a todas las superiores (Charm, TCharm, AMPI, etc...).

El propósito de los algoritmos de balanceo de carga es el de proveer el mejor entorno de ejecución (el conjunto óptimo de procesadores) donde se pueda desplegar la computación. La elección del mejor mapa de procesadores en cada momento supone aplicar estrategias que indiquen qué procesadores son mejores que otros y qué tareas pueden (o deben enviarse a cada procesador de los que componen el nuevo mapa). En cualquier caso estas estrategias necesitan datos objetivos sobre rendimiento. La capa Converse se encarga de mantener estructuras de datos (modificables) donde se almacena información útil sobre perfiles de rendimiento. Por tanto las estrategias de balanceo precisan como información de entrada la relación actual existente entre hebra y procesador y como información de salida ofrece una estructura de datos en la que se debe indicar que hebra irá a cada procesador. Es evidente que la decisión de *que hebra irá a que procesador* es tomada por el propio balanceador. De entre todas las estructuras de datos que se pueden encontrar en el núcleo de la capa Converse nos hemos centrado con especial atención en las estructuras denominadas $PRCS$ y $THRD$. $PRCS$ mantiene toda la información referente a los procesadores y $THRD$ tiene toda la información referente a la ejecución de cada hebra. De entre los campos que define la estructura $PRCS$, hemos tomado información sobre todo de los campos:

- *cmpLoad*: Que contiene la carga de trabajo de *las hebras de nuestra aplicación* en un procesador determinado.

- *bkgLoad*: Carga de trabajo inferida por otras hebras que no pertenecen a nuestra aplicación (background).

- *avbl*: Indicador boolean que establece la disponibilidad de un procesador para ser usado por hebras de nuestra aplicación.

Respecto a la estructura de datos $THRD$:

- *id*: Idenfiticador usado para identificar a cada hebra.
- *Load*: Carga computacional asociada a cada hebra de nuestra aplicación.

Si establecemos que el cluster donde se ejecutará la aplicación está constituido por un número NP de procesadores e identificamos a cada uno de esos NP procesadores usando números enteros $(1, 2, \ldots, NP)$ entonces para conservar la localidad de los datos sin perjudicar el rendimiento de la computación, nuestros algoritmos de balanceo de carga necesitan información sobre qué procesadores de esos NP son útiles para enviar computación, es decir, cuales están disponibles. De este modo se crea un enlace de procesadores (se crea un mapa virtual de procesadores) de acuerdo a la distribución de los datos. De este modo para cualquier procesador disponible, los algoritmos RakeLP y SlidingLP necesitan conocer el identificador (ID) de los procesadores *disponibles* anterior y posterior. La razón para mantener esta lista de enlaces entre procesadores es que los balanceadores pueden retirar de la computación (marcar como no deseable) a los procesadores cuya carga de trabajo (background load) exceda un umbral máximo. El procedimiento denominado $LinkProcs$ (algoritmo 4.2) se ha creado precisamente para que los balanceadores puedan desactivar o activar procesadores sin perjuicio para la localidad de los datos. Este método es responsable de construir y mantener un lista circular doblemente enlazada con todos los procesadores disponibles (virtualmente enlazados). También se encarga de actualizar el contador de procesadores disponibles ($numAvail$). $LinkProcs$ se apoya en la información provista por la estructura de datos $PRCS$. Además de la estructura descrita en la figura 4.5 se precisa que las hebras se puedan mantener ordenadas para que durante las migraciones no se rompa la dependencia de los datos. Para alcanzar este fin se ha provisto a los balanceadores de una estructura FIFO (se puede observar en la figura 4.5) para migrar hebras entre procesadores con *identificadores consecutivos* de manera ordenada. La citada estructura permite que las hebras que comparten datos (hebras vecinas) se encuentren siempre lo más cerca posible.

Una vez que los balanceadores han identificado y reestablecido la lista de procesadores disponibles (procedimiento $LinkProcs$ se calcula la carga media del sistema (parámetro $AvLoad$). Este valor se computa acumulando toda la carga en cada procesador disponible (carga que

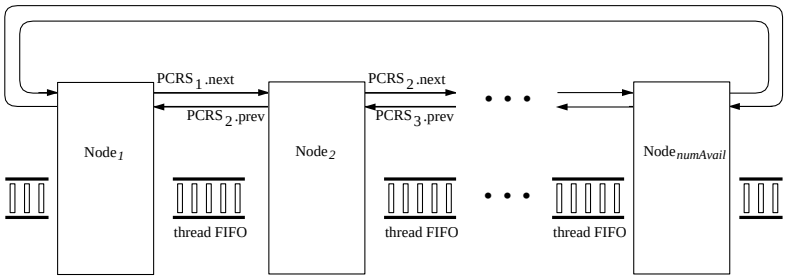


Figura 4.5: Esbozo de la estructura de datos implementada como medio para mantener la localidad de los datos en algoritmos de balanceo dinámico de carga. Las estructuras implementadas para las hebras aseguran una migración ordenada.

Algoritmo 4.2 : $LinkProcs(PRCS)$.

```
1.  first = 0; last = 0; numAvail = 0;
2.  for i = 1 to NP                          bucle: examinar procesadores
3.    PRCS_i.next = PRCS_i;
4.    PRCS_i.prev = PRCS_i;
5.    if (PRCS_i.avbl = TRUE)                comprobar disponibilidad
6.      numAvail = numAvail + 1;
7.      if (last = 0)
8.        first = i                          primer procesador de lista
9.      else                                 otro procesador de la lista
10.       PRCS_last.next = PRCS_i;
11.       PRCS_i.prev = PRCS_last;
12.     last = i;
13. PRCS_last.next = PRCS_first;             Cerrar lista doblemente enlazada
14. PRCS_first.prev = PRCS_last;
15. return numAvail;
```

hace referencia tanto a la carga de nuestras hebras como a la posible carga inducida por hebras de otras aplicaciones *background load*). El algoritmo 4.3 que implementa el procedimiento (*AvgResdLoad*) describe la forma en que se calcula *AvLoad*. En este procedimiento también se calcula un parámetro llamado *Resd* que es el resto obtenido al calcular *AvLoad*. Todas las medidas realizadas sobre la carga computacional se llevan a cabo siempre en relación a la hebra de menor carga. De este modo, parámetros como *cmpLoad*, *bkgLoad*, *AvLoad* y *Resd* son valores *enteros*. El parámetro *NThreads* representa al número de hebras que usa nuestra aplicación, o la aplicación en curso que será balanceada.

El proceso de migración se restringe a las hebras que pertenecen a una aplicación particular. Los procesadores cuya carga de background sobrepasa la carga media del sistema (*AvLoad*) se

Algoritmo 4.3 : *AvgResdLoad.*

1. $TotalLoad := 0$;
2. **for** $i = 1$ **to** $NThreads$
3. $\quad TotalLoad = TotalLoad + THRD_i.Load$;
4. **for** $i = 1$ **to** NP
5. $\quad$ **if** $(PRCS_i.avbl = TRUE)$
6. $\quad\quad TotalLoad = TotalLoad + PRCS_i.bkgLoad$;
7. $AvLoad = TotalLoad/numAvail$;
8. $Resd = TotalLoad$ mód $numAvail$;

consideran procesadores no útiles para la computación y por tanto son marcados como procesadores *no disponibles* ($PRCS_i.avbl = FALSE$), de este modo se consigue que ninguna hebra sea migrada hacia ese procesador y que todas las hebras de ese procesador sean *purgadas* de ese procesador y enviadas (todas) al siguiente procesador en la lista. Esta *operación de limpieza* implica que se han de re–calcular parámetros como $AvLoad$, $Resd$ y $numAvail$. Llegados a este punto, tanto la estrategia RakeLP como la estrategia SlidingLP se encuentran en su punto de partida.

4.4.1. Algoritmo RakeLP

RakeLP es un balanceador centralizado. Esto quiere decir que de todos los procesadores que participan en la computación, sólo uno será el encargado de evaluar el progreso de los demás y de decidir qué hebras quedarán en sus procesadores y cuales serán migradas y dónde. La ejecución del algoritmo de balanceo comienza cuando todas las hebras invocan a la función *MPI_Migrate()* que actúa a modo de *barrera de sincronización*. Tras esta invocación todos los procesadores envían la información del rendimiento de sus hebras al procesador encargado de generar el nuevo mapa hebras–procesadores. El procesador central tras recibir los perfiles de rendimiento de los $numAvail$ procesadores, inicia la estrategia de balanceo. Tras la ejecución de la estrategia, RakeLP, ha de devolver una estructura de datos donde se especifique el conjunto de hebras que serán migradas y el procesador de destino para cada hebra. El runtime actuará de acuerdo al contenido de la estructura de datos haciendo que las hebras cambien de procesador. Cuando cada hebra regrese de su invocación a *MPI_Migrate()* se encontrará en su nuevo procesador y podrá continuar con la ejecución.

El algoritmo RakeLP asegura la convergencia (situación homogénea de carga) se asegura en $numAvail$ iteraciones, es decir, tantas iteraciones como procesadores disponibles. Estas

$numAvail$ iteraciones se llevan a cabo en dos etapas. La primera consiste en $numAvail - Resd$ iteraciones y la segunda, en $Resd$ iteraciones, donde $Resd$ es la carga residual.

Durante las primeras $numAvail - Resd$ iteraciones, RakeLP analiza los procesadores cuya carga está por encima de la media ($AvLoad$) y determina el conjunto de hebras que deberían abandonar el procesador para dejar la carga por debajo de la media. El procesador al que se moverá la carga es el siguiente en la lista virtual de procesadores, por ejemplo, si el procesador i se encuentra sobrecargado entonces la carga excedente (hebras excedentes) serían migradas al procesador $PRCS_i.next$. Para cada iteración se recalcula el total de la carga en cada procesador, información que queda almacenada en $Burd$.

En cada iteración, el balanceador tiene la capacidad de *expulsar* de un procesador un número de hebras determinado, este número se encuentra en la variable $numThreads$ y hace referencia al número de hebras que la aplicación que está siendo balanceada tiene en ese procesador. Cabe notar que el número de hebras de una aplicación viene dado por $NThreads = \sum_{i=1}^{numAvail} numThreads_i$.

El segundo paso del algoritmo RakeLP consiste en iterar $Resd$ veces. Este segundo paso únicamente involucra a aquellos procesadores cuya carga supera la media ($AvLoad$) más un umbral que viene dado por $\Delta Load$. El valor de este umbral ($\Delta Load$),determinado experimentalmente, equivale a la carga de la hebra más ligera.

Los dos pasos fundamentales de RakeLP se describen en el Algoritmo 4.4. En este algoritmo se puede comprobar cómo de un procesador cargado se puede extraer ($deAssign$) una hebra identificada como $THRD.id$, y llevarla o asignarla ($Assign$) a otro procesador cuando se den las condiciones de sobrecarga en el procesador origen.

4.4.2. Algoritmo SlidingLP

El algoritmo SlidingLP también se ha implementado siguiente una estrategia centralizada. Este algoritmo permite a un procesador intercambiar hebras con el procesador sucesor, en contraste con RakeLP donde el flujo de intercambio ha sido unicamente undireccional y desde el procesador i al procesador $i + 1$. En esta estrategia $AvLoad$ se usa para calcular los siguientes parámetros básicos en cada uno de los procesadores disponibles.

- *LoadAdd*: Carga acumulada desde el procesador 1 hasta el procesador actual.
- *LoadEnd*: Valor que debería tener *LoadAdd* para ser una carga balanceada.

Algoritmo 4.4 : RakeLP

```
1.  for i = 1 to numAvail − Resd                    Primera fase: numAvail − Resd iterations
2.    for j = 1 to NP                               Computa total load como
3.      Burd_j = PRCS_j.bkgLoad + PRCS_j.cmpLoad;   bg load + carga de hebras
4.    endfor
5.    for j = 1 to NP
6.      while (Burd_j > AvLoad & numThreads_j > 0)  mientras proc sobrecargado
7.        deAssign(THRD.id, PRCS_j);                retirar hebras preservando la localidad
8.        Assign(THRD.id, PRCS_j.next);             hacia el sig. procesador
9.        numThreads_j = numThreads_j − 1;          actualizar #threads
10.       Burd_j = Burd_j − THRD.cmpLoad;           actualizar load
11.     endwhile
12.   endfor
13. endfor
14. for i = 1 to Resd                               Segunda fase: Resd iteraciones
15.   for j = 1 to NP
16.     Burd_j = PRCS_j.bkgLoad + PRCS_j.cmpLoad;
17.   endfor
18.   for j = 1 to NP
19.     while (Burd_j > AvLoad + ΔLoad &numThreads_j > 0)   nota ΔLoad
20.       deAssign(THRD.id, PRCS_j);
21.       Assign(THRD.id, PRCS_j.next);
22.       numThreads_j = numThreads_j − 1;
23.       Burd_j = Burd_j − THRD.cmpLoad;
24.     endwhile                                    Durante migraciones, métodos Assign y Deassign
25.   endfor                                        junto con LinkProcs y la estructura FIFO
26. endfor                                          desarrollada, mantienen la localidad para las hebras
```

- $Transfer$: Determina la diferencia entre la carga acumulada inicial ($LoadAdd$) y su valor balanceado ($LoadEnd$). Este parámetro indica la cantidad de carga que ha de ser transferida desde el procesador i a $PCRS_i.next$ en el caso de que $Transfer < 0$ o desde el procesador $PCRS_i.next$ hacia el procesador i cuando $Transfer > 0$.

Igual que en la estrategia RakeLP, la computación ajena a nuestra aplicación se cataloga como carga de *background* o *hebras no migrables*. Este hecho establece un límite superior para las hebras que pueden ser expulsadas de un procesador. Igual que en RakeLP, $numThreads$ es el número máximo de hebras de los que un procesador puede deshacerse en cada iteración (tras cada migración este valor se actualiza). Si un procesador requiere deshacerse de más hebras de las que indica el valor $numThreads$, es decir que $|Transfer| > numThreads$, entonces *todas* las hebras de la aplicación han de ser migradas de ese procesador. Sin embargo, la diferencia entre el número de transferencias realizadas y el número de transferencias necesarias ($Transfer - numThreads$) se conservan como transferencias pendientes, aplicables en la siguiente iteración, ya que es posible que este procesador reciba carga de los vecinos.

Algoritmo 4.5 : SlidingLP

```
1.  d = 0; prev = 1;
2.  for i = 1 to NP
3.    if (PRCS_i.avbl = TRUE)                          bucle: calcular carga
4.      d = d + 1;
5.      LoadAdd_i = PCRS_i.bkgLoad + PCRS_i.cmpLoad;
6.      if (d > 1)
7.        LoadAdd_i = LoadAdd_i + LoadAdd_prev;
8.      LoadEnd_i = d × AvLoad;
9.      if (d ≤ Resd)
10.       LoadEnd_i = LoadEnd_i + d;
11.     else
12.       LoadEnd_i = LoadEnd_i + Resd;
13.     Transfer_i = LoadEnd_i − LoadAdd_i;
14.   prev = i;
15. for i = 1 to NP                                    bucle: planear nuevas migraciones
16.   if (PRCS_i.avbl = TRUE)
17.     nxtp = PCRS_i.next
18.     if (Transfer_i > 0 & numThreads_nxtp > 0)
19.       migrLoad = mín(Transfer_i, numThreads_nxtp);
20.       while (migrLoad ≠ 0)
21.         deAssign(THRD.id, PRCS_i.next);
22.         Assign(THRD.id, PRCS_i);
23.         migrLoad = migrLoad − THRD.Load;
24.         id = id + 1;
25.     if (Transfer_i < 0 & numThreads_i > 0
26.       migrLoad = mín(|Transfer_i|, numThreads_i);
27.       while (migrLoad ≠ 0)
28.         deAssign(THRD.id, PRCS_i);
29.         Assign(THRD.id, PRCS_i.next);
30.         migrLoad = migrLoad − THRD.Load;
31.         id = id + 1;
```

El Algoritmo 4.5 contiene una descripción detallada del algoritmo SlidingLP. Nótese que la variable $numThreads_{nxtp}$ especifica el número de hebras que posee el siguiente procesador en la lista, $nxtp = PCRS_i.next$. Aunque no está expresamente indicado es preciso ejecutar el algoritmo de balanceo en más de una ocasión para asegurarse de que todas las transferencias quedan satisfechas.

4.5. Evaluación de las estrategias de balanceo

En esta sección se evalúan ambas estrategias de balanceo dinámico complementadas con las características de conservación de la localidad (RakeLP y SlidingLP).Antes de poder aplicar las estrategias de balanceo a nuestros algoritmos de reconstrucción es deseable poder conocer cómo se comportan con aplicaciones de test. De nuevo, para la aplicación de test hemos elegido nuestra implementación Jacobi3D [206], algoritmo iterativo que afecta a grupos de regiones en una malla. En cada iteración cada región de la malla intercambia parte de su información con sus seis vecinos inmediatos en la malla 3D. A su vez cada región realiza cálculos con los datos que recibe de sus vecinos. Jacobi es un ejemplo clásico de un método numérico iterativo usado para resolver ecuaciones diferenciales parciales. La versión 3D de Jacobi ha sido empleada para este test porque muestra fuertes dependencias entre los datos y enfatiza la necesidad de preservar la localidad de los mismos en aplicaciones como BICAV. Una característica útil e importante en nuestra implementación de Jacobi es el número de hebras en el que se divide la aplicación ya que al incrementar el número de hebras se incrementa también la carga computacional y se obtienen resultados más precisos, también igual que en nuestra aplicación BICAV.

Parte de la evaluación consiste en una comparación de cómo se comporta la aplicación frente a estrategias de balanceo de similares características pero donde la conservación de la localidad de los datos no se ha tenido en cuenta. Para la prueba con la aplicación Jacobi3D se ha empleado como estrategia con la que contrastar a un balanceador de carga estándar que nosotros referiremos como NLPLB (Non-Locality Preserving Load Balancing).

La estrategia de RakeLP es *menos costosa* que la ejecución de la estrategia de SlidingLP (SlidingLP mantiene más estructuras de datos y su complejidad es mayor a la de RakeLP lo que se traduce en mayor tiempo que han de estar las hebras paradas), es por esto que NLPLB usa una estrategia similar a RakeLP.

NLPLB crea dos grupos diferenciados de procesadores: *procesadores cargados* y *procesadores*

ligeros. Estas estructuras son dos listas ordenadas. En el caso de *procesadores cargados* el orden es descendente, el procesador con más carga se encuentra al inicio de la lista. En el caso de *procesadores ligeros* es el procesador con menos carga el que se encuentra en el inicio. De este modo la estrategia traslada hebras del tope de *procesadores cargados* al tope de *procesadores ligeros* hasta que el elemento que ocupa esa posición en *procesadores ligeros* excede el límite de carga (*AvLoad*) para dejar de ser considerado como procesador ligero. El balanceador elimina hebras del procesador más cargado mientras la carga exceda *AvLoad* y las traslada al procesador menos cargado. Este algoritmo no reviste carga computacional apenas (al igual que Rake y en contraste con Sliding) y además consigue un equilibrio de carga cercano al ideal. Es obvio que el principal inconveniente de esta estrategia es que no contempla las condiciones de localidad de los datos, las hebras no son seleccionadas en el procesador de origen y tampoco se contempla que el procesador de destino contenga hebras que compartan datos con la hebra migrada.

Esta prueba de rendimiento ha sido efectuada en un cluster dedicado, compuesto por diez nodos Pentium Xeon (2.4 GHz, 1 GB RAM, 40 GB HDD) donde estaba instalado el kernel de linux *Linux 2.4.22-1.2115.nptlsmp*. Los nodos están interconectados via Gbit Ethernet y, como front-end, el cluster posee un nodo servidor basado en el procesador Xeon 2.66 GHz with 2 GB RAM and 2 $\times$ 73, GB SCSI-320 HDD.

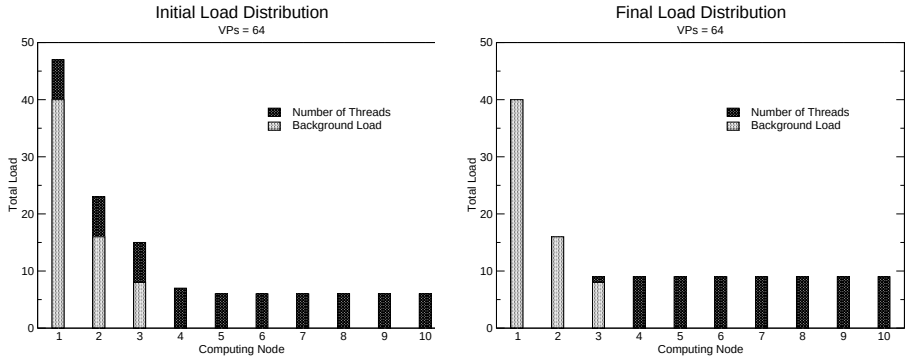
La tabla 4.2 muestra los patrones de carga empleados para testear los balanceadores. La carga (background load) ha sido inducida lanzando otras copias de Jacobi3D sin aplicar balanceo de carga. De este modo hemos podido controlar en cada caso el número de hebras de carga y el número de hebras no migrables (o de background load) que interrumpe nuestra computación. De este modo se pueden simular entornos multiusuario y cuantificar la carga que los mismos inducen. Los nodos 1, 2 y 3 han sido cargados con un número de hebras de background diferente. La tabla 4.2 muestra para cada nodo tanto las hebras de carga (*lthr*) como las hebras de background (*bkgl*) que tiene cada nodo. Como se puede ver en la tabla, inicialmente, las hebras de nuestra aplicación se distribuyen uniformemente entre los nodos del cluster.

El patrón de carga empleado para evaluar las estrategias obliga a que sólo los tres primeros nodos esteń descompensados debido a las hebras extra inducidas en ellos, por esto la carga de cada uno de estos procesadores sobrepasa la carga media del sistema, *AvLoad*. Estos nodos (1,2 y 3) serán marcados como nodos no deseables por nuestras estrategias. De acuerdo a la tabla 4.2, la carga de background que tiene el nodo 1 sobrepasa la media siempre. Consequentemente

Tabla 4.2: Distribución de hebras para el estudio de estrategias (n. significa Nodo)

NThread	n.1		n.2		n.3		n.4	n.5-6	n.7-8	n.9-10
	bkgl	*lthr*	*bkgl*	*lthr*	*bkgl*	*lthr*	*lthr*	*lthr*	*lthr*	*lthr*
8	40	1	2	1	2	1	1	1	1	0
16	40	2	4	2	4	2	2	2	1	1
32	40	4	6	4	6	3	3	3	3	3
64	40	7	16	7	8	7	7	6	6	6
128	80	13	32	13	16	13	13	13	13	12
256	160	26	32	26	16	26	26	26	25	25
512	240	52	32	52	16	51	51	51	51	51

permanecerá marcado como procesador no útil para nuestra computación. En el caso del nodo 2, que este procesador sea marcado como procesador no útil dependerá del tamaño del problema. El nodo 3 tiene una carga de background inferior a la media.

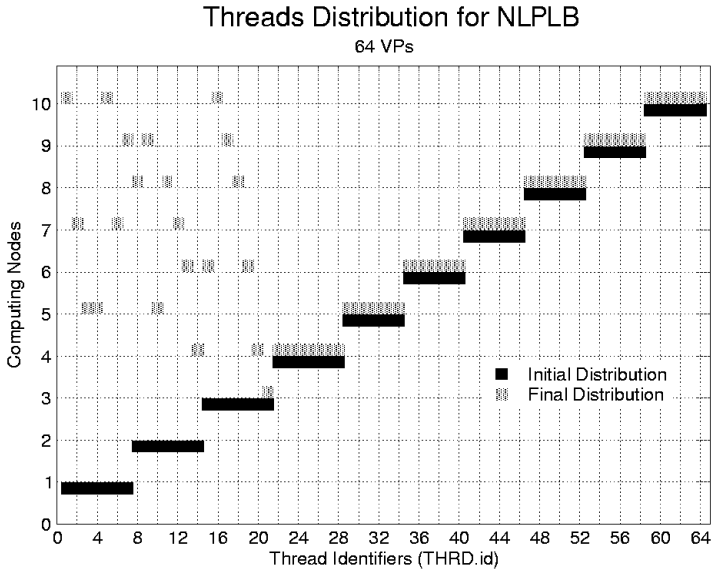


(a) Distribución inicial. (b) Distribución final

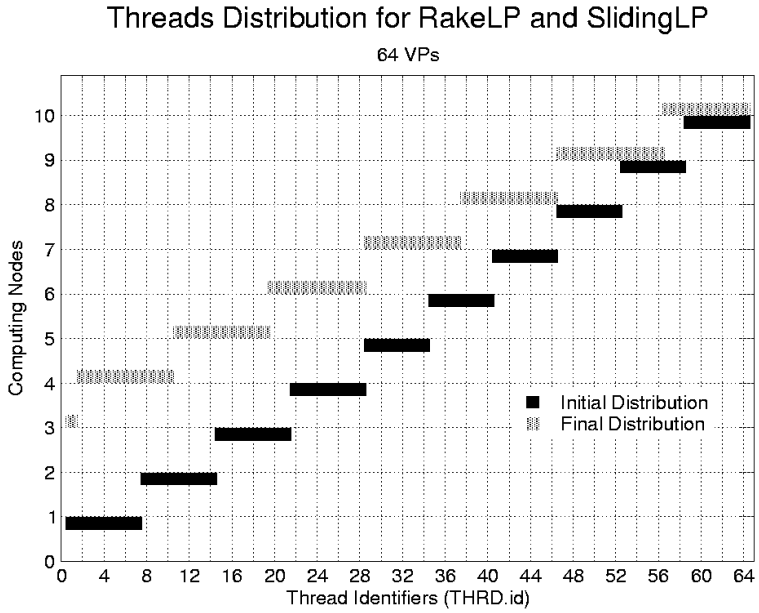
Figura 4.6: Distribución de carga inicial (izquierda) y final (derecha) tras aplicar RakeLP y SlidingLP.

La figura 4.6(a) muestra la distribución inicial de las hebras de carga y de background de acuerdo a lo especificado en la tabla 4.2 para el caso en el que $NThreads = 64$. En la figura 4.6(b) se muestra la distribución de las hebras tras el balanceo. Puede verse cómo afecta la carga de *background* a la decisión de los balanceadores para mover hebras de nuestra aplicación y balancear así la carga. En este caso, los nodos 1 y 2 se han marcado como no útiles debido a que ambos exceden el valor impuesto por *AvLoad*. Las hebras de los nodos 1 y 2, antes de iniciar la estrategia, se mueven (realmente no se migran, sino que se computan a efectos de la estrategia) al siguiente procesador disponible. El movimiento real de las hebras se lleva a cabo una vez se haya terminado la estrategia.

La figura 4.7 pretende mostrar como las estrategias RakeLP y SlidingLP conservan la lo-



(a) Estrategia NLPLB



(b) Estrategias Rake y Sliding

Figura 4.7: Distribución de hebras antes y despues de aplicar las estrategias de balanceo.

calidad de los datos, (ver figura 4.7(b)) mientras que NLPLB migra hebras sin considerar las restricciones de vecindad de las mismas (ver figura 4.7(a)). Estas figuras describen la distribución de las hebras de nuestra aplicación Jacobi3D antes y despues de la aplicación de la estrategia. En concreto la figura 4.7 muestra la actuación de los balanceadores cuando la aplicación tiene 64 hebras ($NThreads = 64$). El identificador de las hebras se representa en el eje X mientras que los nodos usados por las hebras se representan en el eje Y. De este modo, la figura 4.7 describe la distribución final de las hebras en los nodos, es decir en qué nodo se coloca cada hebra. La figura 4.7 muestra como, partiendo de una situación inicial donde cada nodo tenía igual número de hebras de nuestra aplicación, las estrategias tanto RakeLP como SlidingLP consiguen el equilibrio en el balanceo de la carga y además mantienen la relación de vecindad de las hebras de acuerdo a la distribución inicial de la carga. Sin embargo NLPLB, consigue balancear la carga pero sin respetar la distribución de datos original, como se puede apreciar en la figura 4.7(a), por ejemplo, el nodo 7 alberga a las hebras con identificadores $THRD.id = 2, 6, 12, 41$–46, el nodo 10 alberga a las hebras $THRD.id = 1, 5, 16, 59$-64, etc....

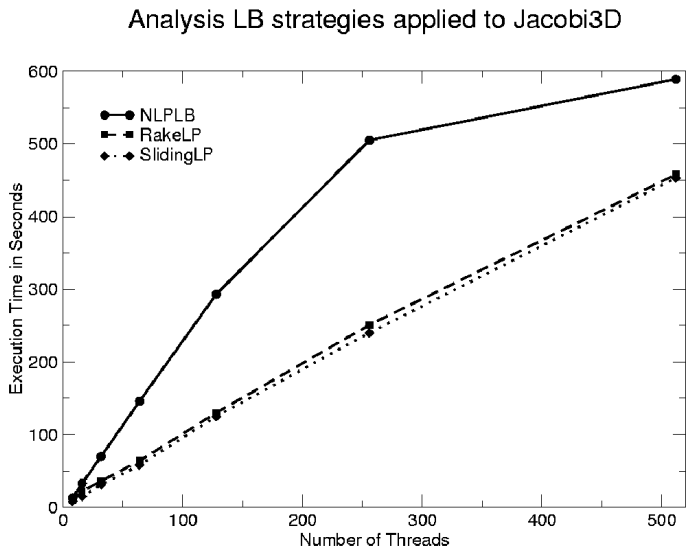


Figura 4.8: Tiempo de ejecución para Jacobi3D usando NLPLB, RakeLP y SlidingLP.

La figura 4.8 muestra los tiempos de ejecución de la aplicación de Jacobi3D para un número creciente de hebras (lo que implica también incremento en la cantidad de trabajo y también de precisión en los resultados). Como se puede apreciar RakeLP y SlidingLP obtienen mejores resultados que cuando se emplea la estrategia NLPLB. Esta última empeora por el incremento de las comunicaciones al separar hebras que deben compartir información. No obstante la estrategia

que mejor escalabilidad presenta es la estrategia RakeLP, como se puede observar en la figura 4.8 para un número de hebras inferior a *256* ambas se comportan igual. No obstante a partir de *256* hebras el rendimiento con RakeLP experimenta una mejoría significativa. De esta serie de tests y de pruebas iniciales que corroboran esta afirmación se ha escogido a RakeLP como estrategia para trabajar con BICAV. Además de que para un número de hebras desde *2* hasta *256* se pueden tomar estos resultados como cota inferior. Es decir las mejoras que obtenemos con este algoritmo usando de *2* a *256* hebras son mejoradas *levemente* por la estrategia SlidingLP, pero no compensa esta mejoría si al incrementar el número de hebras perdemos escalabilidad.

4.6. Adaptabilidad en BICAV

Ya se ha dicho que las estrategias de balanceo no suelen tener en cuenta la naturaleza de la distribución de los datos que impone el programa. Es posible, incluso, afirmar que existen tantas estrategias de balanceo de la carga como problemas necesiten ser balanceados. También es cierto que se pueden emplear estrategias genéricas para conseguir un equilibrio no óptimo pero aceptable que mantenga las cotas de rendimiento en márgenes útiles. El hecho de trabajar con un framework que aporta su propia gestión de las hebras es un punto a favor en el desarrollo de las estrategias.

El modelo de ejecución de BICAV se ha convertido a un modelo de ejecución basado en procesadores virtuales (capítulo 3). Cada hebra posee su propio ID, que es equivalente al *rank* en aplicaciones MPI. Este *rank* es útil para establecer una relación de orden entre ellas y también para identificar a los datos sobre los que trabajar.

La estrategia con la que se evaluará BICAV es RakeLP, estrategia centralizada, que se contrastará con otras estrategias de uso genérico como Greedy [201] y NLBLP que también serán aplicadas al problema de la reconstrucción 3D.

En la implementación de la estrategia Greedy que hemos usado para evaluar nuestra estrategia (RakeLP) se debe tener en cuenta que la re–asociación *hebra–procesador* se lleva a cabo sin tener consideración previa alguna. Esta estrategia se apoya en una estructura de datos *heap* para los procesadores, de modo que el procesador con menos actividad (menos cargado) se encuentra en el tope de este *heap*.Cada vez que se invoca a esta estrategia, este heap se rellena con todos los procesadores por lo que todos los procesadores están, inicialmente, vacíos de carga. La estrategia Greedy también cuenta con un heap para las hebras, la hebra que más tiempo ha estado

en CPU se coloca la primera en este heap. La estrategia sigue con la ejecución de un bucle en el que la hebra que más carga supone se asigna al procesador que menos carga tiene (es decir, se asocian los extremos de cada heap). Tras esta operación la carga del procesador se actualiza y se reordena el heap de los procesadores. La estrategia termina cuando todas las hebras han sido asignadas. En cada invocación de la estrategia se reasigna por completo todo el trabajo.

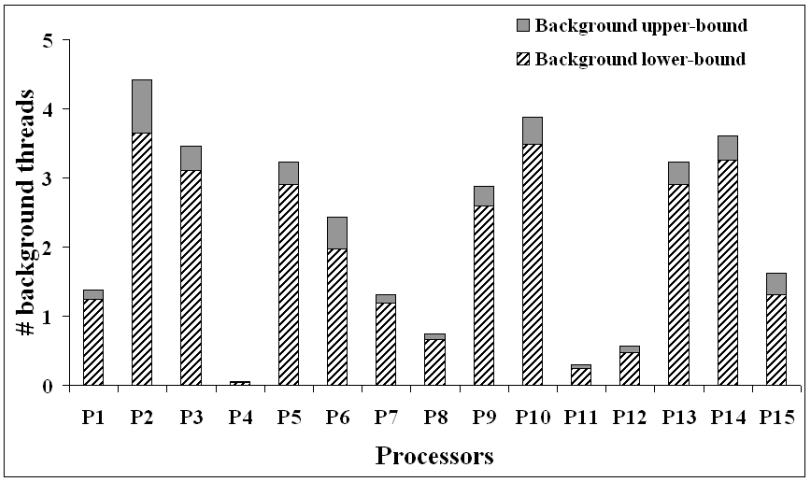
NLBLP que ya fue presentada anteriormente, se puede describir como una estrategia que mejora ostensiblemente la estrategia Greedy pues realiza un balanceo progresivo de la distribución *hebra–procesador* existente, sin ser tan agresivo. NLBLP es una estrategia que ajusta incrementalmente la distribución de las hebras. Para conseguir este refinamiento progresivo, la estrategia necesita de un cierto valor de tolerancia que en nuestros experimentos se ha configurado como 1.02. En este caso esta estrategia emplea dos *heaps*, uno para $\text{Procesadores}_{sobreMedia}$ y otra para los $\text{Procesadores}_{bajoMedia}$. De modo que siempre podemos tener acceso eficiente y rápido al procesador más cargado y al más ocioso. Esta estrategia pasará hebras de un procesador a otro hasta que estén equilibrados. La complejidad de este algoritmo es baja pues sólo se examinan los procesadores sobrecargados.

4.6.1. Evaluación de BICAV con las estrategias de balanceo

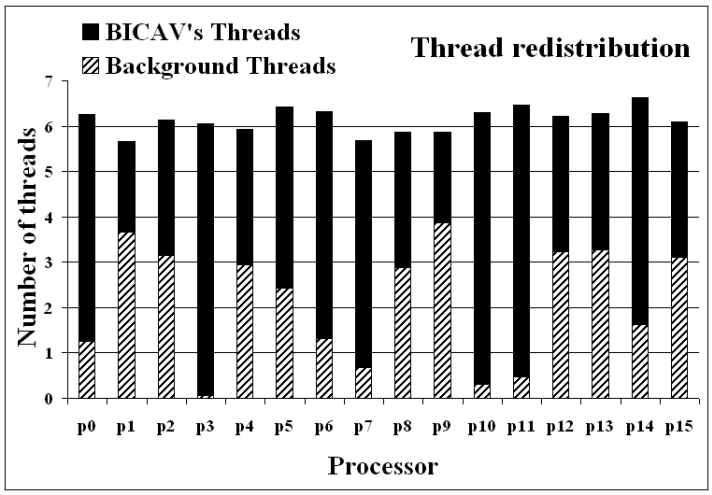
Las dos características que destacan a RakeLP son por un lado la capacidad de conservar la localidad en la distribución de los datos y por otro lado se destaca a la minimización de las latencias. En esta sección se presenta el estudio que analiza el comportamiento de BICAV enlazado con nuestra estrategia RakeLP, con la estrategia Greedy (que representa el máximo exponente de la estrategia genérica) y de NLPLB, estrategia que realiza el equilibrio de la carga respetando –en la medida de lo posible– la distribución inicial de las hebras, pero sin poner atención a las dependencias sobre los datos. BICAV hace gala de fuertes dependencias de datos y por tanto es una aplicación que puede explotar al máximo la capacidad de RakeLP para conservar la localidad en las migraciones de carga, capacidad que se verá reflejada en ganancias de rendimiento para la aplicación.

La frecuencia con la que se invoca al balanceador de carga es un parámetro muy importante y que ha de ser calibrado tras algunos experimentos previos. En esta sección se invocará al balanceador una única vez durante toda la vida de la aplicación. Evidentemente se parte de una situación desfavorable donde las cargas de trabajo inducidas por otros usuarios o la inducida

por la propia aplicación lastrarán mucho a la aplicación en sí.



(a) Valores de background (max y min) por procesador



(b) Emplazamiento de las hebras tras aplicar NLBLP y RakeLP

Figura 4.9: Distribución de background y emplazamiento final de la carga tras aplicar el balanceo.

Los escenarios de carga han sido creados usando otra aplicación que entorpezca el avance de la aplicación de reconstrucción (al igual que en el caso de test) para así crear un desbalanceo controlado. La figura 4.9(a) muestra el número de hebras de background que fueron colocadas en cada procesador. Para cada experimento, *maxback* representa el número máximo de hebras de background, este parámetro depende del número de hebras de la aplicación de reconstrucción por procesador, es decir, $NThreads \div numAvail$. La carga de background se asigna de manera aleatoria a cada procesador, teniendo un valor mínimo de cero y un valor máximo de *maxback*, ver figura 4.9(a).Inicialmente las hebras de la aplicación de reconstrucción se distribuyen por

igual entre todos los procesadores.

La figura 4.9 muestra cómo reaccionan las estrategias ante los escenarios de desbalanceos. Tal y como se discutió en el capítulo anterior, gracias al modelo de objetos concurrentes existen ganancias en rendimiento al dar cabida a más de un *flujo de control* en el mismo procesador. El número máximo de hebras simultáneas en un procesador depende de la aplicación y del procesador, no obstante la estrategia de balanceo no debería exceder este número para no lastrar el rendimiento. De excederlo, el rendimiento en ese procesador caería y en la próxima invocación al balanceador deberá corregirse esta actuación. Tener este dato en cuenta hará que entre las dos invocaciones, la aplicación no vea dilapidado su rendimiento.

De acuerdo a los experimentos realizados en el capítulo anterior, a partir de 16 procesadores BICAV muestra divergencias significativas entre sus dos implementaciones (AMPI y MPI). El número de hebras escogido por procesador debe encontrarse entre 4 y 8. De modo que para testear las estrategias de balanceo con BICAV se escogerá como configuración 16 procesadores y 64 hebras de aplicación ($NThreads = 64$).

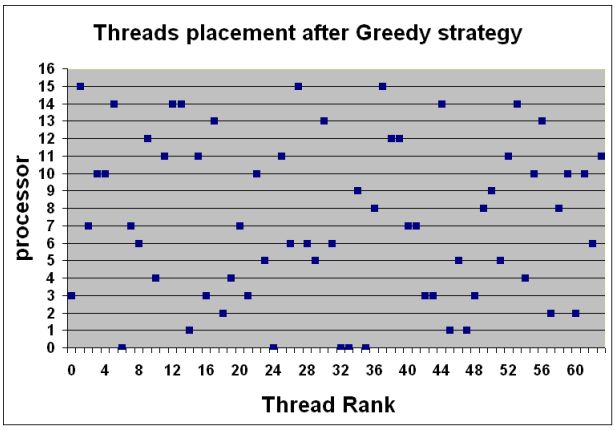
Tras la migración, ver figura 4.9(b), RakeLP y NLBLP reaccionaron de manera similar, es decir, migrando únicamente las hebras necesarias sin mover un alto porcentaje de las hebras de la aplicación. Sin embargo, la decisión adoptada por la estrategia Greedy no siguió la misma tendencia establecida por RakeLP y NLBLP, en cuanto al número de hebras desplazadas, aunque su distribución diverge sólo ligeramente respecto a la de RakeLP y NLBLP. Lo que se puede deducir de estas figuras es que las tres estrategias reaccionaron de manera similar –en cuanto al número de hebras final por procesador– al efecto de la carga de background. No obstante hay un aspecto clave, *dependencia de los datos*.

Lo que se especifica en la figura 4.10 puede contrastarse formalmente empleando la desviación estándar de la carga total del sistema, normalizada a la carga media, σ (ver ecuación 4.1).

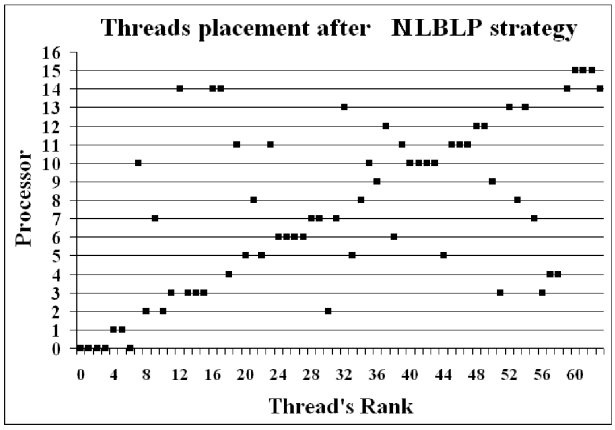
$$\sigma = \frac{1}{\mu} \cdot \sqrt{\frac{\sum_{i=1}^{N}(l_i - \mu)^2}{N}} \tag{4.1}$$

Donde l_i es la carga de cada nodo en el sistema y μ es la carga media de todo el sistema.

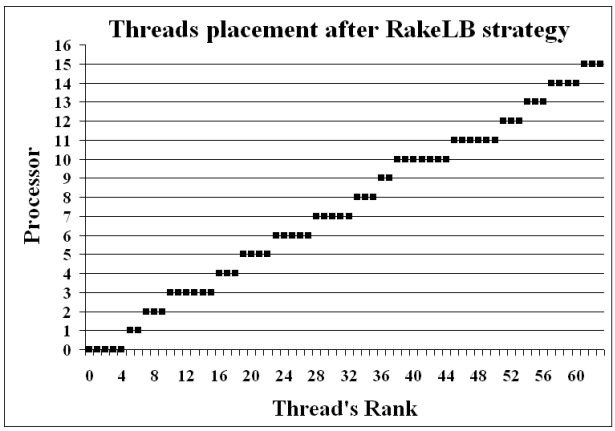
El valor inicial que toma σ supera el 0,5 donde 1,0 significa total desorden y desbalanceo. Tras aplicar las estrategias de balanceo, todas las estrategias dieron como resultado un sistema casi totalmente balanceado (valores inferiores a 0,049 para Greedy e inferiores a 0,04 para RakeLP y



(a) Distribución tras la estrategia Greedy



(b) Distribución tras la estrategia Refine



(c) Distribución tras la estrategia RakeLP

Figura 4.10: Emplazamiento final de las hebras

Tabla 4.3: Walltime ratio y overlap ratio para BICAV

	Greedy	NLPLB	Rake
Cputime / Walltime ratio	0.34	0.52	0.70
Master's Walltime (% relativo a Greedy)	100	66.59	50.29

NLPLB). No obstante y a pesar del buen trabajo realizado por Greedy, el número de migraciones a efectuar es numeroso y se pierde mucho tiempo. Con NLBLP el número de migraciones a efectuar se reduce, ganando así tiempo de estrategia pero no se respeta el orden de distribución de los datos por lo que la comunicación entre las hebras se verá incrementada innecesariamente a través de la red. El punto clave explotado por RakeLP es justamente la *dependencia de los datos*.

En la figura 4.10(c), puede verse que tras la aplicación de la estrategia se alcanza una distribución de la carga casi homogénea (σ es casi cero). Se aprecia al contrastar con la figura 4.10(a) y con la figura 4.10(b) que a pesar de que el valor analítico σ es casi idéntico, únicamente RakeLP considera la localidad de los datos.

Mantener en un procesador el número adecuado de hebras es una ventaja indiscutible para la aplicación, pero si además se respeta cómo estas hebras llevan a cabo su computación como hace RakeLP entonces las mejoras se ven notablemente subrayadas. Si se observa la tabla 4.3 se puede observar que usando RakeLP la ejecución de BICAV en el cluster prácticamente se reduce a la mitad del tiempo empleado si se ejecuta la estrategia Greedy ya que Greedy emplea demasiado tiempo en mover casi todas las hebras y además al deshacer la característica de vecindad (respecto a los datos) se incrementan las comunicaciones. Si se compara con NLPLB se puede observar la ganancia real que supone respetar la localidad de los datos. Los datos mostrados en la tabla 4.3 están normalizados respecto al tiempo de ejecución empleado por BICAV usando la estrategia de balanceo Greedy.

Podemos decir que la razón por la que BICAV es más rápido con la estrategia RakeLP se encuentra en la combinación de dos aspectos, por un lado las *latencias* que son inferiores que con cualquiera de las otras estrategias (se posee suficiente computación con la que solapar las comunicaciones) y por otro la conservación en la medida de lo posible de las dependencias entre los datos (lo que refuerza el primer aspecto). Esta capacidad de aprovechar el solape hace que la computación pueda avanzar más que con otras estrategias.

4.7. Invocación Asíncrona de las Estrategias de Balanceo

Las aplicaciones iterativas han de especificar puntos en los que invocar a las estrategias de balanceo de carga. Normalmente se escoge un número prudente de iteraciones de acuerdo a la irregularidad esperada de la ejecución. No hay otra forma en la que podamos intervenir *bajo demanda*. Cada cierto número de iteraciones se invoca al balanceador que recoge todas las estadísticas de rendimiento y las evalúa de acuerdo a nuestra estrategia. El inconveniente de esta técnica es el de la imposición de la barrera de sincronización que supone invocar al balanceador (incluso si el problema ya está balanceado). En la implementación de la reconstrucción donde se ha empleado AMPI, actuar sobre esta política es algo complejo pues se debe modificar el modelo de objetos usado por el compilador. Sin embargo en la versión orientada a objetos, la inclusión de un nuevo objeto no es difícil. La propuesta que se hace es la de incluir un nuevo objeto por cada procesador (o grupo de procesadores) que actúe como un entrenador, es decir, un objeto que sea capaz de detectar pérdidas de rendimiento en la computación asociada a nuestros objetos y que sea capaz, en ese momento, de invocar al balanceador para corregir la situación. Este método funciona de manera similar a como funciona un *equipo de natación* en una *piscina*, donde el nuevo objeto propuesto se encarga de controlar el tiempo que tarda cada objeto en hacer su trabajo, igual que haría un entrenador. Si un nadador lento entra en la calle de un nadador veloz, el nadador veloz verá perjudicados sus tiempos. En nuestro caso la solución a la pérdida de rendimiento está dentro de la estrategia, únicamente hay que ser capaces de invocarla cuando sea apropiado. Volviendo al paradigma del equipo de natación, si nuestros objetos (nadadores) experimentan un decremento en el rendimiento, el entrenador (estrategia) puede decidir intervenir para cambiarlos de calle. A los nadadores (objetos de otras aplicaciones) lentos que han causado la pérdida de rendimiento, probablemente no podamos moverlos pero si podemos intentar evitarlos cambiando de calle. La Figura 4.11 muestra varios casos en los que la computación orientada a objetos puede verse lastrada bien por la intervención de terceros o bien porque unos objetos se encuentren con más carga (o en procesadores más lentos) que otros. Cada calle de las descritas en la imagen pueden considerarse como nodos de un cluster o núcleos de un procesador multicore e incluso las tres *piscinas* podrían llegar a representar el caso de un cluster de multicores.

En la Figura 4.11, (a) representa a una ejecución donde todos los objetos se comportan normalmente y donde no hay necesidad de realizar operaciones de balanceo de carga. Cada vez que

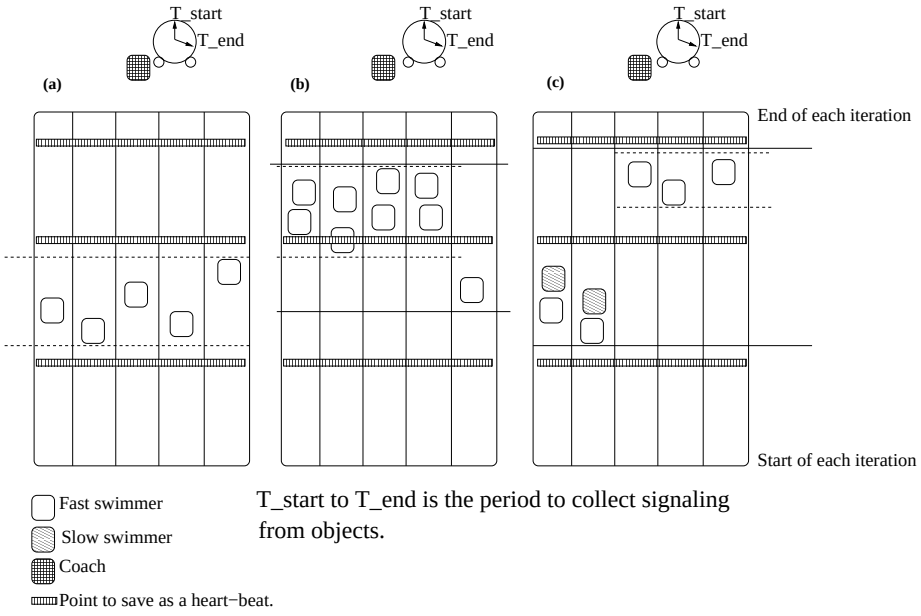


Figura 4.11: Paradigma de balanceo basado en un equipo de natación

un objeto pasa por la *linea rayada* envía un *latido* al entrenador. Es decir, podemos incluir balizas en puntos del código que ejecuta cada objeto donde cada baliza significa el envío de un pequeño mensaje asíncrono al objeto entrenador. Por un lado estos mensajes son fácilmente asumidos y no suponen carga apreciable en el sistema, por otro lado sirven para conocer la *actividad* de cada objeto y poder así realizar un seguimiento sobre el mismo. (b) muestra el caso en el que un objeto está ejecutándose sobre un procesador más lento o puede estar procesando un conjunto de datos más extenso que los demás, en este caso el entrenador (objeto coach) -objeto *auto* en nuestro código- debería invocar al balanceador. (c) identifica el caso donde la computación se ve afectada por objetos de terceros, sería también beneficioso invocar al balanceador. Sin este método durante la ejecución del caso (a) el balanceador habría sido invocado una vez por cada cierto número de iteraciones, innecesariamente. En (b) y (c) la situación de desequilibrio puede aparecer en cualquier momento durante la ejecución, así que no parece rentable implementar las comprobaciones periódicas. Usar al objeto *auto* o si seguimos usando el paradigma del equipo de natación, al entrenador, hace que el balanceo sea más flexible en contraste con la obligación de detenerse cada cierto número de iteraciones. La tabla 4.4 muestra cómo afecta el uso del objeto auto frente a invocar a los balanceadores una sola vez en toda la ejecución (primer caso) o invocar al balanceador cada cincuenta y cada veinte iteraciones (segundo y tercer caso) frente

Tabla 4.4: Walltime ratio para BICAV, método piscina

	Greedy	Refine	Rake
Walltime 1 invocación (% Greedy)	100	66.59	50.29
Walltime 1 cada 50 iteraciones (% Greedy 1 inv.)	101.5	67.4	50.36
Walltime 1 cada 20 iteraciones (% Greedy 1 inv.)	101.62	67.59	50.41
Walltime usando objeto auto (% Greedy 1 inv.)	100.8	66.63	50.3

a no especificar cuando y confiar en la intervención del objeto *auto* (cuarto caso). La actividad del objeto auto puede monitorizarse con la herramienta *PerfVis* que se ha desarrollado para monitorizar en tiempo de ejecución a las aplicaciones de reconstrucción.

La tabla 4.4 muestra los tiempos (walltimes) de la aplicación tomando como referencia el tiempo (walltime) de la aplicación cuando se usa Greedy como estrategia de balanceo en el caso de invocar sólo una vez a la estrategia (Greedy 1 call). En cada llamada a la estrategia, toda la computación se detiene en una barrera, el control es transferido al *runtime* que carga la heurística, recupera las trazas de los objetos y decide si el mapeo objeto-procesador debe alterarse o permanecer como está.

Algoritmo 4.6 Algoritmo del método beat(), implementado en el objeto *auto*

```
1.  trigger ⟵ NumberOfBeatsToCollect
2.  if (--trigger==0)
3.    lapse ← gettime();
4.    if (CollectFirstTime)
5.      auto.getFasterTime(lapse);
6.    endif
7.    if (lapse−idealLapse > ε)
8.      objects[].signalObjectsLBCallRecommended();
9.    endif
10.   trigger ← NumberOfBeatsToCollect ;
11. endif
```

Desde el punto de vista del rendimiento, es buena idea el no abusar de las llamadas al balanceador, nuestro método emplaza un objeto (*auto object*) por core / nodo. Los demás objetos

de nuestra computación únicamente envían un mensaje sencillo y de baja prioridad cada vez que el objeto pasa a través de un *checkpoint* (que es una llamada colocada a conciencia en un punto determinado del objeto). De este modo el objeto *auto* recoge un número predeterminado de latidos, calcula el tiempo en recogerlos y si el intervalo se hace cada vez peor invoca al balanceador de carga. El algoritmo 4.6 refleja el funcionamiento básico del método más significativo del objeto auto.

4.7.1. PerfVis: Monitor de rendimiento

Balancear una aplicación científica e iterativa supone implementar una estrategia especialmente diseñada y ajustada al patrón de ejecución de la misma. En nuestro caso la estrategia mide la carga tomando como referencia al número de hebras (ya que el patrón de carga en nuestra aplicación es homogéneo). Al aplicar el balanceo a nuestra aplicación nos encontramos con dos problemas, el primero de ellos es asegurarnos de que no hay hebras (a pesar de que la carga es homogénea) en un procesador que tardan más (en tiempo de CPU) que las hebras de otro procesador en realizar su trabajo. El segundo problema consiste en encontrar la frecuencia óptima de invocación al balanceador.

Respecto del primer problema, al inicio de cada ejecución – y tras cada invocación al balanceador– el framework registra la potencia de cómputo de cada procesador (usando el código explicado en la sección 4.3.1). Es precisamente esta información la que usamos en nuestros balanceadores para crear *hebras de background* en los procesadores más lentos, de modo que nuestras estrategias no se limiten a contar hebras de la aplicación para así realizar el balanceo. Es cierto que cualquier intervención intermedia puede causar efectos adversos en la computación pues entre invocaciones el *framework* no informa sobre la actividad de terceros. En este sentido *PerfVis* (ver figura 4.12) informa –especialmente para aplicaciones costosas en tiempo– sobre la *salud* de la computación asociada a cada hebra u objeto. *PerfVis* aporta la capacidad de *observar* mientras dura la vida de la aplicación, el progreso de la misma sin causar interferencias en la computación de la misma, ofreciendo así la posibilidad de realizar un análisis non–postmorten de la aplicación y de la efectividad de las estrategias (y del objeto *auto*).

Respecto al segundo problema, encontrar la frecuencia óptima de invocación al sistema de balanceo es un problema real. *PerfVis* es útil para comprobar que todos los objetos o hebras laten por igual y que de este modo el balanceo se está llevando a cabo. *PerfVis* no es más que la

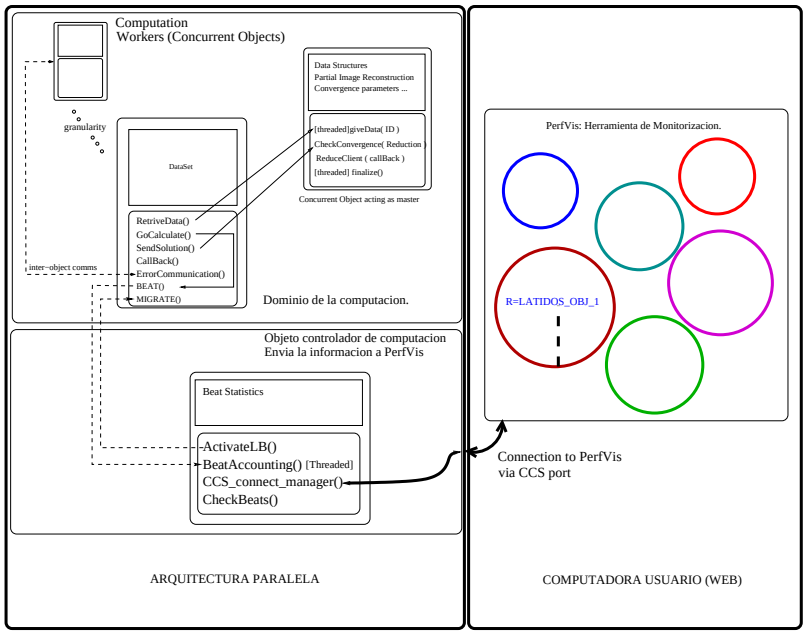


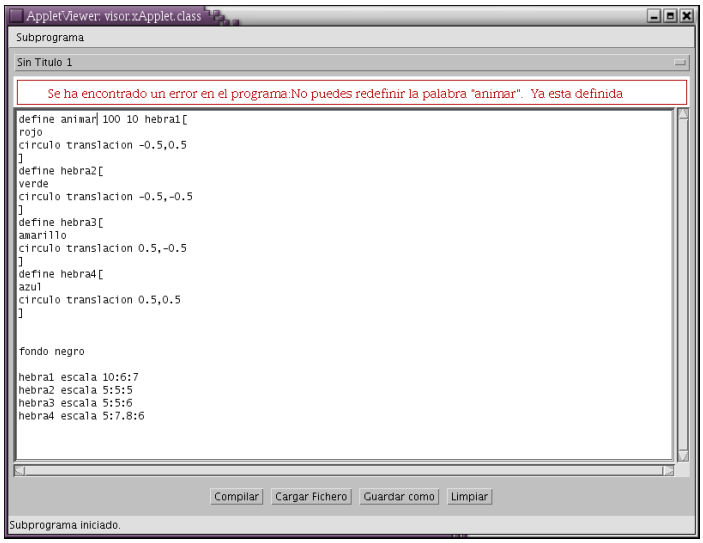
Figura 4.12: Conexión entre el objeto auto y PerfVis. Auto envía el número de latidos / espacio de tiempo que experimenta cada hebra. Esta información se representa con un círculo.

representación gráfica de la actuación de la computación (tengase en cuenta que el acceso a la información –si el trabajo se ejecuta en máquinas paralelas no accesibles– no suele ser sencillo). *PerfVis* muestra la información que recibe –asíncronamente– el objeto *auto* (ver figura 4.12), este objeto (separado en la figura del resto de la computación) se dedica a recoger mensajes que le envían el resto de objetos. Estos mensajes son *latidos*. Cada objeto de la computación (al terminar una iteración, o al enviar un mensaje, o al realizar cualquier actividad que queramos *medir*) envía un mensaje al *objeto auto*. Este objeto (de baja prioridad) recibe el mensaje y lo anota como un latido de ese objeto. Cuando la computación lo permite, el objeto *auto* envía esta información a la interfaz *PerfVis* que analiza sintácticamente y representa gráficamente. Los intervalos de envío de información desde el lado de la computación al lado de la visualización no son regulares ya que se llevan a cabo cuando la computación de los objetos del usuario lo permiten.

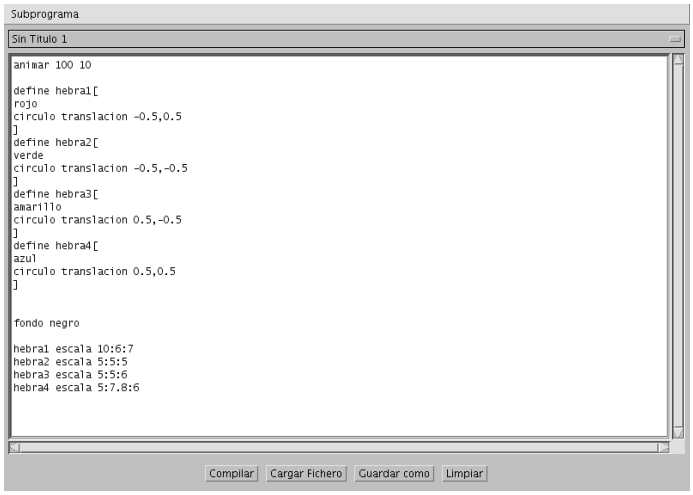
PerVis posee dos partes diferenciadas :

- Analizador sintáctico (ver figura 4.13), que se dedica a recoger la información que recibe desde el objeto *auto* y prevenir posibles errores (figura 4.13(a))–o solapes– que hayan podido ocurrir durante su confección. En este caso, el usuario que recibe la visualización

es advertido y podrá actuar corrigiendo el error (figura 4.13(b)) para proseguir con la visualización.



(a) Datos recibidos con errores



(b) Errores subsanados

Figura 4.13: Analizador

- Representación, es incremental. Cada vez que se recibe información *PerfVis* actualiza en su plantilla de datos el número de latidos de cada objeto y el periodo de segundos durante el que se midió (figura 4.14).

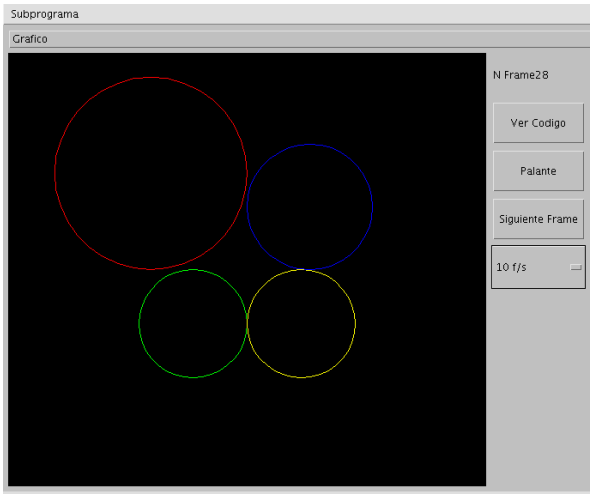


Figura 4.14: Visualizador

4.7.2. Relación con la computación

La herramienta de visualización interpreta en un lenguaje propio, los cortos mensajes de rendimiento que le envía el *objeto auto*. El *objeto auto* se encarga de mantener informado a *PerfVis* –y por ende al usuario de la aplicación– en cada instante sobre cómo se va desarrollando la computación. Indirectamente esta herramienta muestra la carga de trabajo de cada CPU. En la figura 4.15 se puede observar cómo se recogen las estadísticas de cada hebra u objeto y cómo se envían al visor.

4.8. Conclusiones

Las aplicaciones paralelas que se ejecutan en clusters no dedicados, pueden ver dramáticamente afectado su rendimiento debido a desequilibrios inducidos bien por aspectos del problema a tratar (desequilibrios intrínsecos) o por causas externas a la aplicación (desequilibrios extrínsecos). Disponer de técnicas de balanceo de carga para que las aplicaciones puedan adaptarse al entorno computacional disponible es necesario ya que por un lado se consigue usar todos los nodos del cluster por igual y por otro lado, se obtienen ganancias directas en el rendimiento de la aplicación. Pero no todas las estrategias de balanceo son igualmente productivas para todas las aplicaciones. Especialmente si la estrategia debe mostrar ciertos *conocimientos* sobre el patrón de ejecución de la aplicación. En nuestro caso la dependencia que existe entre los datos es parte

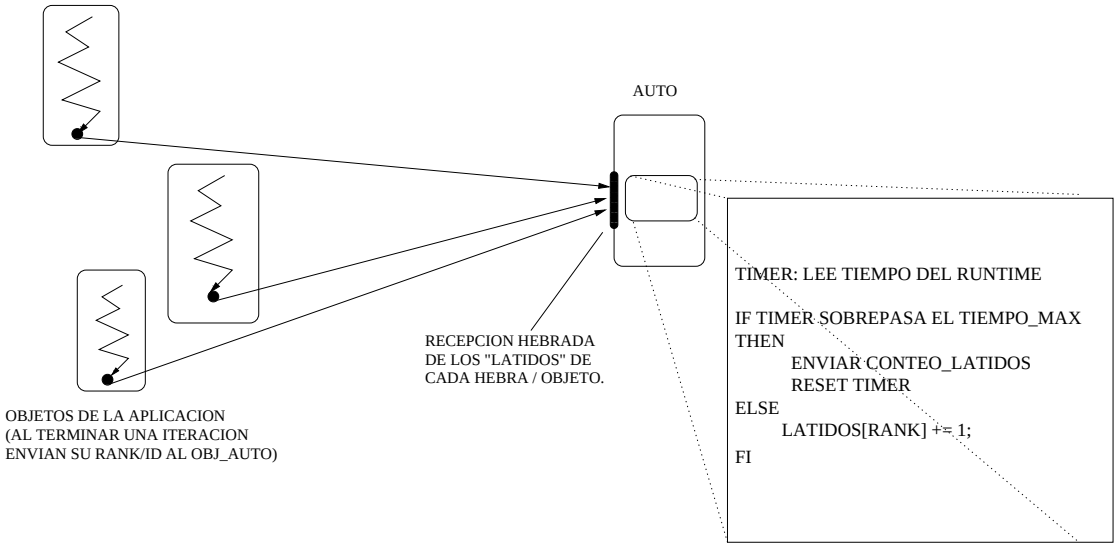


Figura 4.15: Relación de la computación con la herramienta

del problema y es por esto que las estrategias que usemos han de respetar esta característica, por tanto hemos incluido como característica esencial la localidad de los datos como criterio complementario en el balanceo.

Los conceptos, *habilidad para ocultar la latencia* y la *adaptabilidad* han sido estudiados conjuntamente para favorecer la ejecución de aquellas aplicaciones científicas donde existen estrechas relaciones de localidad entre los datos. De este modo la ejecución de tales aplicaciones en clusters o máquinas paralelas compartidas puede llevarse a cabo con resultados notables y favorecedores. Por un lado, el uso de hebras de usuario permite mantener más de un flujo de control por procesador, esta característica junto con los rápidos cambios de contexto que las caracterizan hacen que la labor de ocultar la latencia en las operaciones de comunicación sea sencilla. Por otro lado, las estrategias de balanceo de carga que conservan la localidad de los datos impiden que los procesadores sobrecargados constituyan un cuello de botella en la computación. Además si las estrategias empleadas conservan la localidad en los datos las latencias a ocultar se reducen.

En este capítulo hemos mostrado el trabajo realizado para adaptar y mejorar estrategias de balanceo de carga dinámico para aplicaciones paralelas que se ejecuten clusters no dedicados. Para esto se ha desarrollado una serie de estructuras de datos y de reglas con el fin de conservar la citada localidad de datos durante las migraciones de carga. Este sistema se apoya en circuitos virtuales de procesadores donde se podrán añadir o eliminar procesadores según las condiciones de carga de los mismos. La eficiencia de estas nuevas estrategias de balanceo de carga se ha

demostrado desde varias perspectivas, primero se ha mostrado cualitativamente que mantener la localidad de los datos es conveniente para nuestras aplicaciones y también se ha mostrado cuantitativamente las ganancias obtenidas al balancear la carga teniendo en cuenta factores externos a nuestras aplicaciones y migrando las hebras respetando sus distribuciones de datos.

Capítulo 5

Conclusiones y Líneas de Trabajo Futuro

El trabajo desarrollado en esta de tesis ha consistido proponer nuevos modelos de desarrollo para aplicaciones de elevado coste computacional. Las propuestas se han estudiado sobre aplicaciones del ámbito de la microscopía electrónica de alta resolución, concretamente en aplicaciones diseñadas para reconstruir volúmenes a partir de un conjunto de proyecciones obtenidas desde estos microscopios. Durante todo el trabajo ha estado presente el *aforismo de Nickalus Wirth* donde se considera que un programa no sólo son sus instrucciones sino también los datos sobre los que se opera, de este modo portar aplicaciones como las que se presentan en este campo de trabajo supone no sólo optimizar el algoritmo sino además prestar atención a cómo éste trabaja con los datos asociados. Desde este punto de vista se ha trabajado con el modelo orientado a objetos. La secuencialidad presente en los algoritmos empleados y las arquitecturas sobre las que se ha trabajado invitan a pensar en la programación multihebrada como solución a problemas como el de la ocultación de la latencia. Una de las conclusiones más importantes que se desprende de este trabajo es que se pueden combinar multihebra y orientación a objetos para generar un nuevo espacio de trabajo donde las latencias se puedan ocultar de manera sencilla.

Otro de los problemas abordados en este trabajo ha sido el de la adaptabilidad. Las aplicaciones tradicionales son poco flexibles y una mala distribución inicial del trabajo puede lastrar bastante el resultado de la computación. El modelo de trabajo que se ha adoptado facilita la migración de trabajo entre procesadores y por ende la capacidad de adaptar la computación al entorno sobre el que se lleva a cabo. EL capítulo 4 se encarga de estudiar este aspecto y de proponer estrategias de balanceo para aplicaciones de reconstrucción de imágenes.

El estudio de estas aplicaciones, la optimización y monitorización de las aplicaciones (para

las que hemos creado una aplicación de visualización) hacen que existan numerosas posibilidades para continuar trabajando. Además de estudiar aspectos tan interesantes como el de poder portar la aplicación de reconstrucción a arquitecturas que puedan aparecer en el futuro, lo que hace necesario desarrollar capas de abstracción donde poder ejecutar las aplicaciones.

5.1. Conclusiones y aportaciones de esta tesis

La microscopía electrónica cuenta con dos grandes bloques de algoritmos con los que poder trabajar, los algoritmos de retroproyección filtrada y los algoritmos algebraicos iterativos. La retroproyección filtrada o WBP es una técnica relativamente rápida en las reconstrucciones pero sufren de pérdidas de calidad debido a las condiciones de ruido presentes en las proyecciones con las que se trabaje. Sin embargo, los algoritmos iterativos como ART arrojan mejores resultados con la contrapartida de ser computacionalmente mucho más costosos que WBP, además de ser algoritmos fuertemente secuenciales con lo que la paralelización es dificil de llevar a cabo de manera eficiente.

Las técnicas tradicionales de programación paralela no permiten expresar con flexibilidad la paralelización de estos algoritmos. Los algoritmos de reconstrucción han sido desarrollados tradicionalmente con lenguajes de programación que asumen un único flujo de control por procesador. Estos desarrollos unidos a librerías de paso de mensajes han dado pie a implementaciones donde la eficiencia del algoritmo es función de la destreza del desarrollador. Por otro lado, las diferentes arquitecturas paralelas donde estas aplicaciones son una dificultad añadida.

En el capítulo 3 se han ofrecido alternativas de desarrollo que salvan la dificultad de la paralelización y que limitan la dependencia de la arquitectura. En primer lugar se ha mostrado la efectividad de las abstracciones para el desarrollo de aplicaciones paralelas. Las abstracciones que hemos encontrado útiles han sido la orientación a objetos (especialmente el modelo de Parallel Objects y el uso de objetos concurrentes que explotan la multihebra, es decir Orientación a objetos y multihebra). Tras haber establecido estas dos abstracciones como nuestras herramientas de trabajo, se ha analizado la posibilidad de realizar todo el desarrollo sobre el sistema operativo o sobre un *runtime*. Contar con un *runtime* nos proporciona una capa de abstracción muy útil para poder *obviar a la arquitectura*. Así pues, el primer paso que damos en el capítulo 3 es el desarrollo de nuestro propio *runtime* donde se han creado librerías de sincronización de hebras, donde hemos estudiado aspectos interesantes como los cambios de contexto, el consumo de

memoria, la relación entre el stack y cambios de contexto, la relación entre el stack y la migración a otros procesadores, y colaborar con otro equipo de desarrollo cuyo *runtime* hemos usado. En el capítulo 3 se muestran dos técnicas de desarrollo : una de *grano grueso* donde usando objetos empaquetadores somos capaces de *embutir* al código c tradicional y poder así aprovechar el código ya desarrollado y otra de *grano fino* donde la aplicación de reconstrucción se reescribe usando la orientación a objetos. Se ha demostrado que ambas técnicas son capaces de afrontar la ocultación de latencias de manera sencilla y eficiente.

En el capítulo 4 se ha mostrado cómo proporcionar la capacidad de adaptabilidad a aplicaciones de reconstrucción. Se han desarrollado técnicas de balanceo que estudian el avance de la computación y que proponen nuevas localicaciones para cada uno de los objetos de la misma. Las estrategias desarrolladas tienen en cuenta la distribución de los datos a reconstruir. La ventaja de estas técnicas frente a las existentes es que además de equilibrar la carga, el movimiento de los objetos no incrementa el uso de la red pues pretende emplazar en el mismo procesador a aquellos objetos cuyos conjuntos de datos estén relacionados. En este capítulo se ha mostrado la eficacia de estas técnicas frente a aplicaciones que carecen de la capacidad de adaptar su carga, así como el mejor comportamiento de nuestras estrategias frente a estrategias genéricas existentes.

Las aplicaciones iterativas y adaptativas sufren el inconveniente de tener que especificar un número de iteraciones tras las que se debe detener la computación y analizar el rendimiento de cada objeto para poder balancear la carga si es preciso. Esto hace que si la aplicación no sufre interferencias, se pierda tiempo innecesario en realizar los continuos chequeos. Una de las primeras soluciones ha sido la de optimizar en todo lo posible el algoritmo de balanceo hasta llegar a un orden O(N log N), pero invocar al balanceador supone contar con una barrera de sincronización global con lo que toda la computación se ve sometida al objeto más lento. En la opción de grano grueso, la inclusión de objetos auxiliares que nada tengan que ver con la asociación objeto–código C es muy compleja y sujeta a errores, por lo que en este caso se tiene que decidir inexorablemente el número de iteraciones tras las que se parará el objeto para ser chequeado. Sin embargo en la alternativa de grano fino, incluir objetos extra es una tarea sencilla. En este caso se ha implementado paralelamente a la aplicación de reconstrucción un objeto que controla el avance de los objetos dedicados a la computación, a este paradigma de chequeo lo hemos denominado *swimming pool method*. Este método consiste en que cada objeto

computacional envía un mensaje asíncrono y muy simple al objeto auxiliar, cada vez que pase por determinados puntos de su iteración (estos puntos son definidos por nosotros). El objeto auxiliar recoge un número predeterminado de mensajes y cronometra el tiempo en recogerlos. Cada lapso de tiempo en recoger estos mensajes es controlado y un incremento en los tiempos supone una pérdida de rendimiento y la necesidad de parar la computación para invocar al balanceador. Este método es eficiente pero aún queda mucho trabajo por delante antes de que sea un método plenamente eficaz.

Otro de los problemas asociado a las aplicaciones que se ejecutan en máquinas paralelas es la idoneidad de poder inspeccionar visualmente y en tiempo de ejecución el avance de la computación. En el capítulo 4 se muestra el desarrollo de una aplicación que muestra los latidos de cada objeto que se encuentra en la máquina. La intervención de esta aplicación es mínima pues quien la nutre de información es el propio objeto auxiliar. El objeto auxiliar conoce los *latidos o mensajes* asociados a cada objeto. Cuando el objeto auxiliar (que es un objeto de baja prioridad) consigue tiempo de CPU, hace dos tareas básicas: *conteo de mensajes recibidos* y *envío a través de un puerto* de un mensaje con el número de latidos en ese intervalo de tiempo por cada objeto. La aplicación recoge esa información la *analiza sintácticamente* y recompone una escena que ha de ser dibujada en nuestra aplicación de rendimiento (denominada PerfVis).

5.2. Líneas de trabajo futuro

Las líneas abiertas como consecuencia de este trabajo de tesis son varias, entre las principales se encuentra :

- Runtime : Hemos desarrollado un runtime Java capaz de crear objetos que contienen a procesos escritos en C. La sincronización de estos objetos dentro de la máquina virtual también está desarrollada, pero aún hay que construir una extensión que permita la invocación de métodos en objetos que no están presentes en el mismo procesador que el objeto invocador.

- Modelo de objetos: Para la construcción de la aplicación de grano fino hemos empleado un patrón HLPC (High Level Parallel Construct) tradicional como es el FARM. Crear nuevos patrones HLPC puede ser una forma de limar y perfeccionar el modelo de relación que emplean los objetos entre ellos. Perfeccionar el modelo de invocación asíncrona del

balanceador supondría también una mejora evidente en nuestros desarrollos.

- PerfVis : La herramienta de monitorización únicamente recibe información y la visualiza pero el puerto que emplea para recibir la información desde la máquina paralela admite comunicación en ambos sentidos, es posible que desde la aplicación se puedan alterar parámetros de la reconstrucción en función del estado de los objetos, por ejemplo el número de bloques o la condición de convergencia o el tipo estrategia de balanceo a emplear (si no se observa una alteración notable en los objetos durante un tiempo prudente, podremos cambiar la estrategia que hemos desarrollado por la estrategia NullLB, que es una estrategia de balanceo vacía por lo que si el objeto auxiliar decidiera invocar al balanceador, éste invocaría a una estrategía que no tendría efecto sobre la computación y sus objetos asociados).

- Capas de abstracción : Esta línea de trabajo es, de todas las posibles que se pueden abrir, quizás la más interesante. Una práctica común en aplicaciones hebradas es la de modificar el scheduler del sistema operativo para que nuestras hebras (en nuestra aplicación) sean gestionadas con más eficacia, especialmente ocurre esto con los procesadores multicore simétricos y se está estudiando cómo hacerlo en los futuros procesadores multicore asimétricos. Numerosas publicaciones aventuran que no es una tarea fácil. Por otro lado contar con un *runtime* que se autogestiona sus propias hebras, en la situación descrita, tampoco es muy efectivo. La línea propuesta consiste en desarrollar capas de adaptación a cada máquina. Cada procesador cuenta con numerosos contadores de rendimiento que de poder enviárselos al runtime serían de gran utilidad para afinar aplicaciones.

Capítulo 6

Publicaciones Relacionadas con la Tesis

Durante el desarrollo de este trabajo de tesis, se han realizado publicaciones que recogen los resultados obtenidos. En una primera etapa, el trabajo se centró en la experimentación de técnicas de programación orientadas a objetos como medio para portar aplicaciones científicas a un entorno elemental de balanceo de carga, con el fin de hacer que la computación pudiera hacerse adaptable al entorno. Durante esta primera etapa afrontamos diversos desarrollos entre los que destacaron los objetos concurrentes que ofrece el runitme de charm++. Las publicaciones relacionadas son: *Balanceo dinámico de carga mediante migración de hebras* [232], donde comenzamos a desarrollar un marco de trabajo para aplicaciones científicas, *Migración de objetos concurrentes para balanceo dinámico de carga* [233] y *Concurrent Object Migration Preserving Data Locality* [234]. Concretamente en ésta última publicación sentamos las bases de las estrategías de balanceo que son necesarias en nuestro campo de trabajo, la reconstrucción 3D de imágenes tomográficas.

Con la experiencia de las publicaciones citadas, afrontamos el desarrollo de la primera versión de una aplicación científica donde realizamos el estudio de la ventaja que supone el usar orientación a objetos en la computación basada en clusters. En *Object cluster computing. Advantages of object oriented load balancing strategies* [235] se encuentran los resultados más importantes.

El runtime charm ofrece la posibilidad de crear desarrollos a más bajo nivel, en *Interval Parallel Global Optimization with Charm++* [236], se envió como comunicación los resultados más destacables del desarrollo de una aplicación científica irregular usando objetos concurrentes charm++. El resultado de este estudio y comunicación fue publicado más tarde en [237].

Tras el periodo de estudio del efecto de la orientación a objetos, usando los objetos concurrentes que ofrece charm++, volvimos a perfeccionar nuestras estrategias de balanceo con resultados notables, que presentamos en las siguientes comunicaciones:

* Load balancing strategy using concurrent objects [238].

* Local Load Balancing Strategy Keeping Data Neighborhood [239]

* La Localidad de los Datos, criterio complementario en operaciones de balanceo dinámico de carga [240].

* Data neighboring in local load balancing operations [241].

La serie de resultados obtenidos y la experiencia obtenida con ellos nos permitió afrontar el estudio en profundidad de nuestras aplicaciones de reconstrucción. La lista de trabajos se muestra a continuación.

* Concurrent Threads for Parallel Electron Tomography [242]. Donde se presentan los primeros resultados de aplicar a las aplicaciones de reconstrucción de nuestro grupo, las técnica de paralelización basadas en hebras de usuario en el entorno Charm++ AMPI.

* Multithreaded Tomographic Reconstruction [243]. Ampliación del trabajo anterior donde se evalúa la escalabilidad de la aplicación en un cluster de nuestro grupo de investigación y se sientan las restricciones (número de hebras ideal por procesador, etc) para que la aplicación pueda escalar y ocultar posibles latencias. *Conferencia listada en Core, rank C, conferencia*

(EuroPVM/MPI), 42750 European PVM/MPI Users'Group Conference EuroPVM/MPI C 0805 Distributed Computing.

* Parallel Electron Tomography in a Multithreaded Environment [244]. Estudio de los perfiles de uso de memoria de una aplicación de reconstrucción usando el entorno tradicional MPI - C y de nuestra implementación particular en el entorno multihebrado. Se presentan las principales complicaciones como que las aplicaciones clásicas desarrolladas en éste entorno usan en exceso las variables globales, y cómo se puede resolver esta problemática.

* Latency Hiding and adaptability for parallel iterative algorithms. [245]. En este trabajo, tras obtener los resultados del número de hebras por procesador ideales, de los perfiles de comunicaciones, de la capacidad de conmutación entre hebras; estudiamos cómo mantener por procesador, en cada momento, el número de hebras que permitiera la mejor escalabilidad de la aplicación. Un número de hebras (o VPs) excesivo por procesador es tan malo para el rendimiento como el no tener la capacidad de ocultar la latencia.

* Exploiting Concurrence in 3D Parallel Iterative Reconstruction Algorithms. [246]. La capacidad de combinar estrategias de balanceo diseñadas a medida para la aplicación de reconstrucción y que además sean capaces de migrar respetando los patrones de comunicación y el número de hebras ideal para cada nodo hacen que estemos en condiciones de poder ocultar muy bien la latencia, siempre que los cambios de contexto se manejen con flexibilidad (como es el caso del entorno que estamos usando).

* A Load Balancing Framework in Multithreaded Tomographic Reconstruction [247]. Y su publicación como capítulo de libro en la editorial IOS [248]. En este trabajo presentamos las dos estrategias de balanceo diseñadas para nuestras aplicaciones de reconstrucción y sus resultados ante estrategias de balanceo convencionales en el campo de las aplicaciones científicas. *Conferencia listada en Core, rank C. 43978 Parallel Computing PARCO C 0805 Distributed Computing.*

* Tomographic image reconstruction using abstractions. [249]. El uso de la orientación a objetos ofrece la posibilidad de obtener esquemas de paralelización muy buenos si el modelo orientado a objetos es preciso. En este artículo comenzamos a estudiar el uso de patrones y abstracciones como modelos de desarrollo para incluir en nuestras aplicaciones de reconstrucción. Las abstracciones MAXPAR y FARM (construcciones paralelas de alto nivel) se aplican aqui.

* High level abstractions for improving parallel image reconstruction algorithms [250] Ampliación de los resultados obtenidos en la publicación anterior.

* Latency hiding and adaptability for parallel iterative algorithms [251]. En este trabajo resaltamos que uno de los principales problemas de las aplicaciones científicas es que necesitan de algún medio de sincronización. En la mayoría de ellas este medio es la iteración. Cuando precisamos balancear estas aplicaciones científicas, por norma, se debe elegir un número fijo de iteraciones cada vez en las que detener toda la computación y evaluar el rendimiento para decidir si se balancea o no.

* From structured to object oriented programming in parallel algorithms for 3D image reconstruction [252]. En este trabajo presentamos el desarrollo orientado a objetos y basado en abstracciones que consideramos para las aplicaciones de reconstrucción. *Conferencia listada en Core, rank A, 42782 European Conference on Object Oriented Programming ECOOP A 0803 Computer Software.*

En las siguientes publicaciones se presenta una solución al problema del balanceo que presentamos en [251], de modo que la estrategia de balanceo de la computación sólo sea evaluada cuando exista una situación real de desequilibrio, sin necesidad de tener que consultar cada cierto número de iteraciones. En estas publicaciones presentamos también la etapa inicial del desarrollo de una aplicación de visualización que nos permite saber si las estrategias de balanceo están funcionando bien con este método.

* A Load Balancing Model for 3D HPC Image Reconstruction Applications To Reduce Unnecessary Overheads. Proceedings of the 9th International Meeting in High Performance Computing for Computational Science, (Vecpar 2010) *Conferencia listada en Core, rank B, 43572 International Conference on Vector and Parallel Processing VecPar B 0805 Distributed Computing.*

* A proposed asynchronous object load balancing method for parallel 3D image reconstruction applications [253]. *Conferencia listada en Core, rank B, 43221 International Conference on Algorithms and Architectures for Parallel Processing ICA3PP B 0805 Distributed Computing.*

Bibliografía

[1] M. J. Zaki, Wei Li, and S. Parthasarathy. Customized dynamic load balancing for a network of workstations. In *HPDC '96: Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing*, page 282, Washington, DC, USA, 1996. IEEE Computer Society.

[2] C. Tianzhou, T. Xingsheng, M. Jianliang, J. Lihan, J. Guanjun, and S. Qingsong. Single thread program parallelism with dataflow abstracting thread. *LNCS:Algorithms and Architectures for Parallel Processing*, 6082/2010:11–21, 2010.

[3] K. Davis and J. Striegnitz. Parallel/high-performance object-oriented scientific computing: Today's research, tomorrow's practice. *Lecture Notes Computer Science*, 5475/2009:104–115, 2009.

[4] C. Bo, X. Yun, Y. Jiaoyun, and J. Haitao. A new parallel method of smith-waterman algorithm on a heterogeneous platform. *LNCS:Algorithms and Architectures for Parallel Processing*, 6081/2010:79–90, 2010.

[5] I. Peláez, F. Almeida, and D. González. From xml specifications to parallel programs. *LNCS:Parallel and Distributed Processing and Applications*, 4330/2006:267–277, 2009.

[6] E. Loh. The ideal hpc programming language. *Commun. ACM*, 53(7):42–47, 2010.

[7] M. Blatt and P. Bastian. C++ components describing parallel domain decomposition and communication. *International Journal of Parallel, Emergent and Distributed Systems*, 24:467–477, 2009.

[8] V. Kindratenko, R. Wilhelmson, R. Brunner, T.J. Martinez, and W. Hwu. High-performance computing with accelerators. *Computing in Science and Engineering*, 12:12–16, 2010.

[9] V. S. Pai and S. Adve. Code transformations to improve memory parallelism. In *MICRO 32: Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, pages 147–155, Washington, DC, USA, 1999. IEEE Computer Society.

[10] J. Sampson, R. Gonzalez, J. F. Collard, N.P. Jouppi, M. Schlansker, and B. Calder. Exploiting fine-grained data parallelism with chip multiprocessors and fast barriers. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 235–246, Washington, DC, USA, 2006. IEEE Computer Society.

[11] T. M. Jones, M. F. P. O'Boyle, J. Abella, A. González, and O. Ergin. Exploring the limits of early register release: Exploiting compiler analysis. *ACM Trans. Archit. Code Optim.*, 6(3):1–30, 2009.

[12] D. August, K. Pingali, D. Chiou, R. Sendag, and J. J. Yi. Programming multicores: Do applications programmers need to write explicitly parallel programs? *Micro, IEEE*, 30(3):19 –33, may-june 2010.

[13] T. Chen, T. Xingsheng, M. Jianliang, J. Lihan, J. Guanjun, and S. Qingsong. Single thread program parallelism with dataflow abstracting thread. *LNCS Algorithms and Architectures for Parallel Processing*, 6082/2010:11–21, 2010.

[14] S. V. Adve and H. J. Boehm. Memory models: a case for rethinking parallel languages and hardware. *Commun. ACM*, 53(8):90–101, 2010.

[15] F. Puntigam. Interfaces of active objects with internal concurrency. In *DO21 '09: Proceedings of the 1st International Workshop on Distributed Objects for the 21st Century*, pages 1–5, New York, NY, USA, 2009. ACM.

[16] F. J. Pollack. New microarchitecture challenges in the coming generations of cmos process technologies (keynote address)(abstract only). In *MICRO 32: Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, page 2, Washington, DC, USA, 1999. IEEE Computer Society.

[17] D.A. Patterson. Computer science education in the 21st century. *Commun. ACM*, 49(3):27–30, 2006.

[18] X. Zhou, J. Yang, M. Chrobak, and Y. Zhang. Performance-aware thermal management via task scheduling. *ACM Trans. Archit. Code Optim.*, 7(1):1–31, 2010.

[19] C. Meenderinck and B. Juurlink. (when) will cmps hit the power wall? *Euro-Par 2008 Workshops - Parallel Processing*, 5415/2009:184–193, 2009.

[20] M.D. Hill and M.R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, July 2008.

[21] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's*, 30(3):202–210, January 2005.

[22] L.A. Stein. Challenging the computational metaphor: Implications for how we think. *Cybernetics and Systems*, 30(6):1–35, August 1999.

[23] B. Weissman, B. Gomes, J. Quittek, and M. Holtkamp. Efficient fine-grain thread migration with active threads. *Parallel Processing Symposium, International*, 0:410–414, 1998.

[24] Gengbin Z., L.V. Kale, and O.S. Lawlor. Multiple flows of control in migratable parallel programs. *Parallel Processing Workshops, 2006. ICPP 2006 Workshops. 2006 International Conference on*, pages 10 pp. –444, 0-0 2006.

[25] Huang C., O. Lawlor, and L. V. Kale. Adaptive mpi. In *In Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03*, pages 306–322, 2003.

[26] L. V. Kale and S. Krishnan. Charm++: A portable concurrent object oriented system based on c++. Technical report, Parallel Programming Lab, UIUC, Champaign, IL, USA, 1993.

[27] B. Gregg. Visualizing system latency. *Commun. ACM*, 53(7):48–54, 2010.

[28] M.J. Bridges, N. Vachharajani, Yun Z., T. Jablin, and D.I. August. Revisiting the sequential programming model for the multicore era. *Micro, IEEE*, 28(1):12 –20, jan. 2008.

[29] W. Feng and P. Balaji. Tools and environments for multicore and many-core architectures. *Computer*, 42(12):26 –27, dec. 2009.

[30] R.T. Kouzes, G.A. Anderson, S.T. Elbert, I. Gorton, and D.K. Gracio. The changing paradigm of data-intensive computing. *Computer*, 42(1):26 –34, jan. 2009.

[31] J. Michalakes and M. Vachharajani. Gpu acceleration of numerical weather prediction. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1 –7, apr. 2008.

[32] E.R. Rodrigues, P. O. A. Navaux, J. Panetta, C. L. Mendes, and L. V. Kale. Optimizing an MPI Weather Forecasting Model via Processor Virtualization. In *Proceedings of International Conference on High Performance Computing (HiPC)*, 2010.

[33] H. Garsden and G.F. Lewis. Gravitational microlensing: A parallel, large-data implementation. *New Astronomy*, 15(2):181 – 188, 2010.

[34] Y. Sun and J.R. Cavallaro. Efficient hardware implementation of a highly-parallel 3gpp lte/lte-advance turbo decoder. *Integration, the VLSI Journal*, In Press, Corrected Proof:–, 2010.

[35] H. M. P. Couchman. Simulating the formation of large-scale cosmic structure with particle-grid methods. *Journal of Computational and Applied Mathematics*, 109(1-2):373 – 406, 1999.

[36] B. Parhami. *Computer Architecture. From Microprocessors to Supercomputers.* Oxford University Press, 2005.

[37] J. L. Hennesy and D. A. Patterson. *Computer Architecture. A Quantitative Approach.* Morgan Kaufmann, Elsevier, 4th edition, 2006.

[38] M Oskin. The revolution inside the box. *Commun. ACM*, 51(7):70–78, 2008.

[39] L. Hoffmann. Q & a. *Commun. ACM*, 53(7):112–ff, 2010.

[40] J. Parkhurst, J. Darringer, and B. Grundmann. From single core to multi-core: preparing for a new exponential. In *ICCAD '06: Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, pages 67–72, New York, NY, USA, 2006. ACM.

[41] A Sodan, J Machina, A Deshmeh, K Macnaughton, and B Esbaugh. Parallelism via multithreaded and multicore cpus. *Computer*, PP(99):1 –1, 2010.

[42] R.R. Schaller. Moore's law: past, present and future. *Spectrum, IEEE*, 34(6):52–59, Jun 1997.

[43] S. Hamilton. Taking moore's law into the next century. *Computer*, 32(1):43–48, Jan 1999.

[44] A. Aizcorbe and S. Kortum. Moore's law and the semiconductor industry: A vintage model. *Scandinavian Journal of Economics*, 107(4):603–630, December 2005.

[45] S. Tyagi. Moore's law: A cmos scaling perspective. *Physical and Failure Analysis of Integrated Circuits, 2007. IPFA 2007. 14th International Symposium on the*, pages 10–15, July 2007.

[46] D. Post. The promise of science-based computational engineering. *Computing in Science Engineering*, 11(3):3 –4, may. 2009.

[47] S Schneider, J Yeom, and DÑikolopoulos. Programming multiprocessors with explicitly managed memory hierarchies. *Computer*, PP(99):1 –1, 2009.

[48] R. Buyya. *High Performance Cluster Computing: Programming and Applications.* Prentice Hall, NJ, USA, 1999.

[49] T. Kelly, Y. Wang, S. Lafortune, and S. Mahlke. Eliminating concurrency bugs with control engineering. *Computer*, 42(12):52 –60, dec. 2009.

[50] Yang Zhang and E. Luke. Concurrent composition using loci. *Computing in Science Engineering*, 11(3):27 –35, may. 2009.

[51] J. Falcou. Parallel programming with skeletons. *Computing in Science Engineering*, 11(3):58 –63, may. 2009.

[52] R. Buyya. *High Performance Cluster Computing: Architectures and Systems.* Prentice Hall, NJ, USA, 1999.

[53] M. Madadi, A.C. Jones, C.H. Arns, and M.A. Knackstedt. 3d imaging and simulation of elastic properties of porous materials. *Computing in Science Engineering*, 11(4):65 –73, jul. 2009.

[54] M. Saadatfar. Computer simulation of granular materials. *Computing in Science Engineering*, 11(1):66 –74, jan. 2009.

[55] D. Muller-Wichards and W. Ronsch. Scalability of algorithms: An analytic approach. *Parallel Computing*, 21(6):937–952, June 1995.

[56] D. Moncrieff, R. E. Overill, and S. Wilson. Heterogeneous computing machines and amdahl's law. *Parallel Computing*, 22(3):407–413, March 1996.

[57] E.F. Walker, R. Floyd, and P.Ñeves. Asynchronous remote operation execution in distributed. *10th International Conference on Distributed Computing Systems, Proceedings*, pages 253–259, 1990.

[58] S.C. Goldstein, K.E. Schauser, and D. Culler. Lazy threads: Implementing a fast parallel call. *Journal of Parallel and Distributed Computing*, 37(1):5–20, 1996.

[59] G.M. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. In AFIPS Press, editor, *AFIPS Conference Proceedings*, volume 30, pages 483–485, Atlantic City, N.J., 1967.

[60] J.L. Gustafson. The consequences of fixed time performance measurement. In *System Sciences, 1992. Proceedings of the Twenty-Fifth Hawaii International Conference on*, volume ii, pages 113 –124 vol.2, jan. 1992.

[61] F. Corberá, R. Asenjo, and E. L. Zapata. Progressive shape analysis for real c codes. In *IEEE International Conference on Parallel Processing (ICPP'2001)*, page 373, Valencia, Spain, 2001.

[62] D. Sima, T. Fountain, and P. Kacsuk. *Advanced Computer Architectures: A Design Space Approach.* Pearson Education, 1998.

[63] J. Shen and M. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors.* Mc Graw Hill, 2005.

[64] D. Shelepov, J. C. Saez-Alcaide, S. Jeffery, A. Fedorova, N. Perez, Z. F. Huang, S. Blagodurov, and V. Kumar. Hass: a scheduler for heterogeneous multicore systems. *SIGOPS Operating Systems Review*, 43(2):66–75, 2009.

[65] M.A. Gray. Getting started with gpu programming. *Computing in Science Engineering*, 11(4):61 –64, jul. 2009.

[66] J. Cohen and M. Garland. Novel architectures: Solving computational problems with gpu computing. *Computing in Science Engineering*, 11(5):58 –63, sep. 2009.

[67] S.Ñegara, G. Zheng, K.C. Pan, N.Ñegara, R. Johnson, L. V. Kale, and P. Ricker. Automatic MPI to AMPI Program Transformation using Photran. In *3rd Workshop on Productivity and Performance (PROPER 2010)*, Ischia/Naples/Italy, August 2010.

[68] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in c programs. In *MICRO 40: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 356–369, Washington, DC, USA, 2007. IEEE Computer Society.

[69] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 105–118, Washington, DC, USA, 2005. IEEE Computer Society.

[70] B. Wilkinson and M. Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. Prentice Hall, 2nd edition, 2004.

[71] Daasch W. R. Driscoll M. A. Accurate predictions of parallel program execution time. *Journal of Parallel and Distributed Computing*, 25(1):16–30, February 1995.

[72] S.K. Moore. Multicore is bad news for supercomputers. *Spectrum, IEEE*, 45(11):15 –15, nov. 2008.

[73] X-H. Sun and Y. Chen. Reevaluating amdahl's law in the multicore era. *Journal of Parallel and Distributed Computing*, 70(2):183 – 188, 2010.

[74] B. Hendrickson and J. W. Berry. Graph analysis with high-performance computing. *Computing in Science & Engineering*, 10(2):14–19, 2008.

[75] R. Strawn. High-performance computing for rotorcraft modeling and simulation. *Computing in Science & Engineering*, 12(5):27 –35, 2010.

[76] S. Ellison, J. Dean, M. Johnson, C. Prebola, C. Fabozzi, and A. Cenko. Supplying air warfare capability through high-performance computing. *Computing in Science & Engineering*, 12(5):18 –26, 2010.

[77] P. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann, 1st edition, 1995.

[78] V.S. Sunderam, G.A. Geist, J. Dongarra, and R. Manchek. The pvm concurrent computing system: Evolution, experiences, and trends. *Parallel Computing*, 20(4):531 – 545, 1994. Message Passing Interfaces.

[79] M. Tateno, Y. Hagiwara, and J. Kang. Development and applications of a novel qm/mm hybrid molecular dynamics calculation system on highly parallel supercomputer systems. *Biophysical Journal*, 98(3, Supplement 1):572a – 572a, 2010.

[80] D. Vidal, R. Roy, and F. Bertrand. On improving the performance of large parallel lattice boltzmann flow simulations in heterogeneous porous media. *Computers Fluids*, 39(2):324 – 337, 2010.

[81] D. Vidal, R. Roy, and F. Bertrand. A parallel workload balanced and memory efficient lattice-boltzmann algorithm with single unit bgk relaxation time for laminar newtonian flows. *Computers Fluids*, 39(8):1411 – 1423, 2010.

[82] G. Cong and Bader D.A. Designing irregular parallel algorithms with mutual exclusion and lock-free protocols. *Journal of Parallel and Distributed Computing*, 66(6):854–866, June 2006.

[83] M. Diaz, S. Romero, B. Rubio, E. Soler, and J.M. Troya. Adding aspect-oriented concepts to the high-performance component model of sbasco. In *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*, pages 21 –27, 18-20 2009.

[84] R. Rabbah and K. Palem. Bridging processor and memory performance in ilp processors via data remapping, 2001.

[85] R. Shiveley. Performance scaling in the multi-core era. Publicación técnica Intel Software Network, 30 octubre, 2007.http://softwarecommunity.intel.com/articles/eng/2766.htm.

[86] R.P. Kendall, A. Mark, S.E. Squires, and C.A. Halverson. Condor: Case study of a large-scale, physics-based code development project. *Computing in Science Engineering*, 12(3):22 –27, may-june 2010.

[87] J. A. Board jr, L. V. Kale, K. Schulten, R.D. Skeel, and T. Schlick. Modeling biomolecules: Large sclaes, longer durations. *IEEE Computational Science & Engineering*, 1:19–30, 1994.

[88] B. Lewis and D. J. Berg. *Multithreaded programming with Pthreads*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998.

[89] K. Hinsen. The promises of functional programming. *Computing in Science Engineering*, 11(4):86 –90, july-aug. 2009.

[90] K. Laufer and G.K. Thiruvathukal. Scientific programming: The promises of typed, pure, and lazy functional programming: Part ii. *Computing in Science Engineering*, 11(5):68 –75, sept.-oct. 2009.

[91] I Foster, C. Kesselman, and S. Tuecke. The nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, 37(1):70–82, 1996.

[92] L.V. Kale. The Chare Kernel parallel programming language and system. In *Proceedings of the International Conference on Parallel Processing*, volume II, pages 17–25, August 1990.

[93] L. V. Kale and S. Krishnan. Charm++: a portable concurrent object oriented system based on c++. In *OOPSLA '93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 91–108, New York, NY, USA, 1993. ACM.

[94] E. Thorsten, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *ISCA '92: Proceedings of the 19th annual international symposium on Computer architecture*, pages 256–266, New York, NY, USA, 1992. ACM.

[95] J. V. W. Reynders, P. J. Hinker, J. C. Cummings, S. R. Atlas, S. Banerjee, W. F. Humphrey, S. R. Karmesin, K. Keahey, M. Srikant, and M. D. Tholburn. Pooma: A

framework for scientific simulations of paralllel architectures. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming in C++*, pages 547–588. MIT Press, 1996.

[96] X. Liu. Exploiting object-based parallelism on multi-core multi-processor clusters. In *Parallel and Distributed Computing, Applications and Technologies, 2007. PDCAT '07. Eighth International Conference on*, pages 259 –266, dec. 2007.

[97] J. Duffy. *Concurrent Programming on Windows*. Addison-Wesley Professional, 2008.

[98] T. L. Veldhuizen and M. E. Jernigan. Will c++ be faster than fortran? In *Proceedings of the 1st International Scientific Computing in Object-Oriented Parallel Environments ISCOPE'97*, Berlin, Heidelberg, New York, Tokyo, 1997. Springer-Verlag.

[99] H.W. Schmidt and J. Chen. Reasoning about concurrent objects. *Software Engineering Conference, 1995. Proceedings., 1995 Asia Pacific*, pages 86–95, Dec 1995.

[100] H. Bi. Towards abstraction of message passing programming. In *Advances in Parallel and Distributed Computing, 1997. Proceedings*, pages 100 –107, mar. 1997.

[101] A. A. Chien, J. Dolby, B. Gangul, V. Karamcheti, and X. Zhang. Evaluating high level parallel programming support for irregular applications in icc++. *Softw. Pract. Exper.*, 28(11):1213–1243, 1998.

[102] J. Frank. *Three-Dimensional Electron Microscopy of Macromolecular Assemblies*. Elsevier, 1996.

[103] J. Frank. *Electron Tomography. Three-Dimensional Imaging with the Transmission Electron Microscope*. Plenum Press, New York, ninth dover printing, tenth gpo printing edition, 1992.

[104] M. J. Schillaci. Total tomography [review of advances in discrete tomography and its applications (herman, g.t. and kuba, a., eds.; 2007)]. *Computing in Science Engineering*, 11(2):12 –13, march-april 2009.

[105] A.J. Koster, R. Grimm, D. Typke, R. Hegerl, A. Stoschek, J. Walz, and W. Baumeister. Perspectives of molecular and cellular electron tomography. *Journal of Structural Biology*, 120:276–308, 1997.

[106] B.F. McEwen and M. Marko. The emergence of electron tomography as an important tool for investigating cellular ultrastructure. *J. Histochem. Cytochem.*, 49:553–564, 2001.

[107] K. Grunewald, O. Medalia, A. Gross, A.C. Steven, and W. Baumeister. Prospects of electron cryotomography to visualize macromolecular complexes inside cellular compartments: implications of crowding. *Biophys Chem*, 100:577–591, 2003.

[108] A. Sali, R. Glaeser, T. Earnest, and W. Baumeister. From words to literature in structural proteomics. *Nature*, 422:216–225, 2003.

[109] A.S. Frangakis, J. Bohm, F. Forster, S.Ñickell, D.Ñicastro, D. Typke, R. Hegerl, and W. Baumeister. Identification of macromolecular complexes in cryoelectron tomograms of phantom cells. *Proceedings National Academy of Sciences, USA*, 99:14153–14158, 2002.

[110] G. A. Perkins, C. W. Renken, J. Y. Song, T. G. Frey, S. J. Young, S. Lamont, M. E. Martone, S. Lindsey, and M. H. Ellisman. Electron tomography of large, multicomponent biological structures. *Journal of Structural Biology*, 120(3):219 – 227, 1997.

[111] M.S. Ladinsky, D.N. Mastronarde, J.R. McIntosh, K.E Howell, and L.A. Staehelin. Golgi structure in three dimensions: Functional insights from the normal rat kidney cell. *Journal of Cell Biology*, 144:1135–1149, 1999.

[112] R. Grimm, H. Singh, R. Rachel, D. Typke, and W. Zillig, W.and Baumeister. Electron tomography of ice-embedded prokaryotic cells. *Journal of Biophysics*, 74:1031–1042, 1998.

[113] O. Medalia, I. Weber, A.S. Frangakis, D.Ñicastro, G. Gerisch, and W. Baumeister. Macromolecular architecture in eukaryotic cells visualized by cryoelectron tomography. *Science*, 298:1209–1213, 2002.

[114] P. Penczek, M. Marko, K. Buttle, and J. Frank. Double-tilt electron tomography. *Ultramicroscopy*, 60:393–410, 1995.

[115] D.N. Mastronarde. Dual-axis tomography: An approach with alignment methods that preserve resolution. *Journal of Structural Biology*, 120(3):343 – 352, 1997.

[116] Y. Censor, D. Gordon, and R. Gordon. Component averaging: An efficient iterative parallel algorithm for large and sparse unstructured problems. *Parallel Computing*, 27(6):777 – 808, 2001.

[117] Y. Censor, D. Gordon, and R. Gordon. Bicav: a block-iterative parallel algorithm for sparse systems with pixel-related weighting. *Medical Imaging, IEEE Transactions on*, 20(10):1050–1060, Oct. 2001.

[118] G.T. Herman. *Medical Imaging. Systems, Techniques and Applications*, chapter Algebraic Reconstruction Techniques in Medical Imaging, pages 1–42. Gordon and Breach Science, 1998.

[119] M. Rademacher. *Electron Tomography. Three-Dimensional Imaging with the Transmission Electron Microscope*, chapter Weighted Back-Projection Methods, pages 91–115. Plenum Press, 1992.

[120] R. Marabini, G. T. Herman, and J. M. Carazo. 3d reconstruction in electron microscopy using art with smooth spherically symmetric volume elements (blobs). *Journal of Ultramicroscopy*, 72(1-2):53 – 65, 1998.

[121] J.J. Fernandez, A.F. Lawrence, J. Roca, I. Garcia, M.H. Ellisman, and J.M. Carazo. High performance electron tomography of complex biological specimens. *Journal of Structural Biology.*, 138:6–20, 2002.

[122] R. Marabini, E. Rietzel, E. Schroeder, Herman G.T., and J.M. Carazo. Three-dimensional reconstruction from reduced sets of very noisy images acquired following a single-axis tilt schema: Application of a new three-dimensional reconstruction algorithm and objective comparison with weighted backprojection. *J. Struct. Biol.*, 120:363–371, 1997.

[123] C.O.S. Sorzano, R. Marabini, N. Boisset, E. Rietzel, R. Schroder, G.T. Herman, and J.M. Carazo. The effect of overabundant projection directions on 3D reconstruction algorithms. *Journal of Structural Biology*, 133:108–118, 2001.

[124] R. M. Lewitt. Alternatives to voxels for image representation in iterative reconstruction algorithms. *Physics in Medicine and Biology*, 37(1-2):705 – 716, 1992.

[125] S. Matej, R.M. Lewitt, and G.T. Herman. Practical considerations for 3-D image reconstruction using spherically symmetric volume elements. *IEEE Trans. Med. Imag.*, 15:68–78, 1996.

[126] B. Wilkinson and M. Allen. *Parallel programming: techniques and applications using networked workstations and parallel computers*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1999.

[127] J.J. Fernández, J.M. Carazo, and I. García. Three-dimensional reconstruction of cellular structures by electron microscope tomography and parallel computing. *Journal of Parallel and Distributed Computing*, 64(2):285 – 300, 2004.

[128] J.J. Fernández, D. Gordon, and R. Gordon. Efficient parallel implementation of iterative reconstruction algorithms for electron tomography. *Journal of Parallel and Distributed Computing*, 68(5):626 – 640, 2008.

[129] J.R. Bilbao-Castro, R. Marabini, C.O.S. Sorzano, I. García, J.M. Carazo, and J.J. Fernández. Exploiting desktop supercomputing for three-dimensional electron microscopy reconstructions using art with blobs. *Journal of Structural Biology*, 165(1):19 – 26, 2009.

[130] R. Buyya, editor. *High Performance Cluster Computing, Vols. I and II*. Prentice Hall, 1999.

[131] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1994.

[132] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 2 edition, October 1992.

[133] E.K. Maghraoui, T. Desell, B.K. Szymanski, and C. A. Varela. Dynamic malleability in iterative mpi applications. In *Proceedings of Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2007), Rio de Janeiro, Brazil*, pages 591–598. IEEE Computer Society, May 2007.

[134] V. Pankratius, C. Schaefer, A. Jannesari, and W.F. Tichy. Software engineering for multicore systems: an experience report. In *IWMSE '08: Proceedings of the 1st international workshop on Multicore software engineering*, pages 53–60, New York, NY, USA, 2008. ACM.

[135] Intel. ITRS: International technology roadmap for semiconductors, (edición 2008). http://www.itrs.net, 2008.

[136] X.H. Sun and Y. Chen. Reevaluating amdahl's law in the multicore era. *Journal of Parallel and Distributed Computing*, 70(2):183 – 188, 2010.

[137] E.A. Lee. The problem with threads. *Computer*, 39(5):33 – 42, may. 2006.

[138] A.M. Tyrrell and J.D. Nicoud. Scheduling and parallel operations on the transputer. *Microprocessing and Microprogramming*, 26(3):175 – 185, 1989.

[139] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel Computing*, 22(6):789 – 828, 1996.

[140] B. M. Chapman and L. Huang. Enhancing openmp and its implementation for programming multicore systems. In Christian H. Bischof, H. Martin Bücker, Paul Gibbon, Gerhard R. Joubert, Thomas Lippert, Bernd Mohr, and Frans J. Peters, editors, *Parallel Computing: Architectures, Algorithms and Applications, ParCo 2007, Forschungszentrum Jülich and RWTH Aachen University, Germany, 4-7 September 2007*, Advances in Parallel Computing, pages 3–18. IOS Press, 2007.

[141] O. Sievert and H. Casanova. A simple mpi process swapping architecture for iterative applications. *International Journal of High Performance Computing Applications*, 18(3):341, 2004.

[142] K. El Maghraoui, B.K. Szymanski, and C. Varela. An architecture for reconfigurable iterative mpi applications in dynamic environments. In *Proc. of the Sixth International Conference on Parallel Processing and Applied Mathematics (PPAM 2005)*, pages 258–271. Springer Verlag, 2005.

[143] C. Huang, O. S. Lawlor, and L. V. Kalé. Adaptive mpi. In Lawrence Rauchwerger, editor, *LCPC*, volume 2958 of *Lecture Notes in Computer Science*, pages 306–322. Springer, 2003.

[144] L. V. Kalé and S. Krishnan. Charm++: A portable concurrent object oriented system based on c++. In *OOPSLA*, pages 91–108, 1993.

[145] M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Casçaval. How much parallelism is there in irregular applications? In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 3–14, New York, NY, USA, 2009. ACM.

[146] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[147] G. Burns, R. Daoud, and J. Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[148] J. M. Squyres and A. Lumsdaine. A Component Architecture for LAM/MPI. In *Proceedings, 10th European PVM/MPI Users'Group Meeting*, number 2840 in Lecture Notes in Computer Science, pages 379–387, Venice, Italy, September / October 2003. Springer-Verlag.

[149] E. Gabriel, G.E. Fagg, T. Bosilca, G.and Angskun, J.J. Dongarra, J.M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R.H. Castain, D.J. Daniel, R.L. Graham, and T.S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users'Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[150] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 211–222, New York, NY, USA, 2007. ACM.

[151] B.C. McCandless and A. Lumsdaine. The role of abstraction in high-performance computing. *Lecture Notes in Computer Science*, 1343/1997:203–210, 1997.

[152] J. A. Álvarez and J. Roca-Piera. High level abstractions for improving parallel image reconstruction algorithms. In Sigeru Omatu, Miguel Rocha, José Bravo, Florentino Fernández Riverola, Emilio Corchado, Andrés Bustillo, and Juan M. Corchado, editors, *IWANN (2)*, volume 5518 of *Lecture Notes in Computer Science*, pages 50–57. Springer, 2009.

[153] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[154] J. Whitney. Living in a multi-core world: Tips for developers. http://developer.amd.com/pages/621200628.aspx.

[155] H. Sutter. Use threads correctly = isolation + asynchronous messages. *Dr Dobb's*, 6:7–15, 2009.

[156] H. Sutter. Break-up and interleave work to keep threads responsive. *Dr Dobb's*, 6(5):13–26, 2009.

[157] A. Corradi and L. Leonardi. Parallelism in object-oriented programming languages. In *International Conference on Computer Languages*, pages 271–280, 1990.

[158] A. W. Roscoe and C. A. R. Hoare. The laws of occam programming. *Theoretical Computer Science*, 60(2):177 – 229, 1988.

[159] N. Francez, C. A. R. Hoare, D.J. Lehmann, and W.P. De Roever. Semantics of nondeterminism, concurrency, and communication. *Journal of Computer and System Sciences*, 19(3):290 – 308, 1979.

[160] G. Hilderink, J. Broenink, W. Vervoort, and A. Bakkers. Communicating Java Threads. In A. Bakkers, editor, *Parallel Programming and Java, Proceedings of WoTUG 20*, volume 50, pages 48–76, University of Twente, Netherlands, 1997. IOS Press, Netherlands.

[161] N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for c#. *ACM Transactions on Programming Languages and Systems*, 26(5):769–804, 2004.

[162] A. Ciampolini, A. Corradi, and L. Leonardi. The support for a dynamic parallel object model on a transputer-based architecture. In *Tenth Annual International Phoenix Conference on Computers and Communications*, pages 316–322, 1991.

[163] M. Boari, A. Ciampolini, and A. Corradi. Programming environments for transputer-based architectures. In *CompEuro'91. 5th Annual European Computer Conference on Advanced Computer Technology, Reliable Systems and Applications.*, pages 94–102, 1991.

[164] F. Zambonelli, M. Pugassi, L. Leonardi, and N. Scarabottolo. Experiences on porting a parallel objects environment from atransputer network to a pvm-based system. In *Proceedings of the Fourth Euromicro Workshop on Parallel and Distributed Processing, 1996. PDP'96.*, pages 367–374, 1996.

[165] A. Corradi and L. Leonardi. Po: an object model to express parallelism. *SIGPLAN Notices*, 24(4):152–155, 1989.

[166] W. Gropp, E. Lusk, R. Ross, and R. Thakur. Using mpi-2: Advanced features of the message passing interface. *Cluster Computing, IEEE International Conference on*, 0:19, 2003.

[167] M. Papathomas. Concurrency in object-oriented programming languages. In *Object-Oriented Software Composition*, pages 31–68. Prentice Hall, 1995.

[168] A. Radenski. Object-oriented programming and parallelism. *Information Science.*, 93(1):1–7, 1996.

[169] W. Joosen, S. Bijnens, and P. Verbaeten. Massively parallel programming using object parallelism. In *Programming Models for Massively Parallel Computers.*, pages 144–150, Sep 1993.

[170] J. R. Mühlbacher. Using object-oriented concepts for programming with light-weight processes. *Journal of Microcomputer Applications*, 14(3):263 – 292, 1991.

[171] B. Lewis and D.J. Berg. *Multithreaded programming with Pthreads.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998.

[172] J. A. Alvarez, J. Roca-Piera, and J. J. Fernandez. A load balancing framework in multithreaded tomographic reconstruction. In Christian H. Bischof, H. Martin Bücker, Paul Gibbon, Gerhard R. Joubert, Thomas Lippert, Bernd Mohr, and Frans J. Peters, editors, *PARCO*, volume 15 of *Advances in Parallel Computing*, pages 165–172. IOS Press, 2007.

[173] H. Tang, K. Shen, and T. Yang. Compile/run-time support for threaded mpi execution on multiprogrammed shared memory machines. *SIGPLAN Not.*, 34(8):107–118, 1999.

[174] Leiserson C. E. The case for a concurrency platform. http://www.ddj.com/hpc-high-performance-computing/212001540.

[175] D. K. Lowenthal, V. W. Freeh, and G. R. Andrews. Using fine-grain threads and run-time decision making in parallel computing. *Journal of Parallel Distributed Computing*, 37(1):41–54, 1996.

[176] C. Huang, O. Lawlor, and L. V. Kalé. Adaptive mpi. In *In Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03*, pages 306–322, 2003.

[177] E.K. Maghraoui, T. J. Desell, B. K. Szymanski, and C. A. Varela. The internet operating system: Middleware for adaptive distributed computing. *International Journal of High Performance Computing Applications*, 20(4):467–480, 2006.

[178] D. Keppel. Tools and techniques for building fast portable threads packages. Technical Report UWCSE 93-05-06, University of Washington Department of Computer Science and Engineering, May 1993.

[179] G. Zheng, C. Huang, and L. V. Kalé. Performance evaluation of automatic checkpoint-based fault tolerance for ampi and charm++. *SIGOPS Oper. Syst. Rev.*, 40(2):90–99, 2006.

[180] M. Oskin. The revolution inside the box. *Communications of the ACM*, 51(7):70–78, 2008.

[181] C. Huang, G. Zheng, and Kalé L.V. Supporting adaptivity in mpi for dynamic parallel applications. Technical Report 07-08, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, 2007.

[182] L. V. Kalé, S. Kumar, and K. Varadarajan. A framework for collective personalized communication. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, pages 69–71, Washington, DC, USA, 2003. IEEE Computer Society.

[183] S. Kumar and L. V. Kalé. Scaling collective multicast on fat-tree networks. In *ICPADS*, Newport Beach, CA, July 2004.

[184] C.W. Lee, C. Mendes, and Kalé L.V. Towards Scalable Performance Analysis and Visualization through Data Reduction. In *13th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, Miami, Florida, USA, April 2008.

[185] F. Bassetti, K. Davis, M. V. Marathe, D. J. Quinlan, and B. Philip. Improving cache utilization of linear relaxation methods: Theory and practice. In Satoshi Matsuoka, R. R. Oldehoeft, and Marydell Tholburn, editors, *ISCOPE*, volume 1732 of *Lecture Notes in Computer Science*, pages 25–36. Springer, 1999.

[186] L. Klaus-Peter. Object-oriented concurrent programming. In Raimund K. Ege, Madhu S. Singh, and Bertrand Meyer, editors, *TOOLS (8)*, page 274. Prentice Hall, 1992.

[187] W. D. Clinger. Foundations of actor semantics. Technical report, M.I.T., Cambridge, MA, USA, 1981.

[188] M. Rossainz-López and M. I. Capel-Tunón. An approach to structured parallel programming based on a composition. In *CONIELECOMP*, page 42. IEEE Computer Society, 2006.

[189] A. Corradi and L. Leonardi. Concurrency within objects: layered approach. *Information and Software Technology*, 33(6):403–412, 1991.

[190] B. Cantrill and J. Bonwick. Real-world concurrency. *Commun. ACM*, 51(11):34–39, 2008.

[191] R.G. Lavender and D.C. Schmidt. *Active object: An object behavioral pattern for concurrent programming.* Number 11 in Addison-Wesley. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1996.

[192] T. L. Veldhuizen and M. E. Jernigan. Will c++ be faster than fortran? In *Proceedings of the 1st International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'97)*, Lecture Notes in Computer Science. Springer-Verlag, 1997.

[193] M. Chao, G. Zheng, F. Gioachin, and L.V. Kalé. Optimizing a Parallel Runtime System for Multicore Clusters: A Case Study. In *TeraGrid'10*, Pittsburgh, PA, USA, August 2010.

[194] Matchy J. M. M., Cho-Li W., and F.C.M. Lau. Jessica: Java-enabled single-system-image computing architecture. *Journal of Parallel and Distributed Computing*, 60(10):1194–1222, 2000.

[195] D. Draheim, C. Lutteroth, and G. Weber. Generative programming for c#. *SIGPLAN Not.*, 40(8):29–33, 2005.

[196] R. Chaube, R. L. Carino, and I. Banicescu. Effectiveness of a dynamic load balancing library for scientific applications. In *ISPDC '07: Proceedings of the Sixth International Symposium on Parallel and Distributed Computing*, page 32, Washington, DC, USA, 2007. IEEE Computer Society.

[197] J.C. Chen, G.X. Liao, Jr.S. Hsie, and C.H. Liao. A study of the contribution made by evolutionary learning on dynamic load-balancing problems in distributed computing systems. *Expert Syst. Appl.*, 34(1):357–365, 2008.

[198] H. Gould and J. Tobochnik. *An Introduction to Computer Simulation Methods: Applications to Physical Systems, 3rd Ed.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[199] M. Sonka, V. Hlavac, and R. Boyle. *Image Processing, Analysis, and Machine Vision.* Thomson-Engineering, 2007.

[200] A. Kak and M. Slaney. *Principles of Computerized Tomographic Imaging.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2001.

[201] G. Aggarwal, R. Motwani, and A. Zhu. The load rebalancing problem. *J. Algorithms*, 60(1):42–59, 2006.

[202] K. D. Devine, E. G. Boman, R. T. Heaphy, B. A. Hendrickson, J. D. Teresco, J. Faik, J. E. Flaherty, and L. G. Gervasio. New challenges in dynamic load balancing. *Appl. Numer. Math.*, 52(2–3):133–152, 2005.

[203] C. Fonlupt, P. Marquet, and J. Dekeyser. Data-parallel load balancing strategies. *Parallel Computing*, 24:1665–1684, 1996.

[204] M. Bhandarkar, M. Bh, L. V. Kale, E. de Sturler, and J. Hoeflinger. Object-based adaptive load balancing for mpi programs. In *Proceedings of the International Conference on Computational Science, San Francisco CA, LNCS 2074*, pages 108–117, 2001.

[205] J. Roca, J. C. Ortega, J. A. Álvarez, and J. Mateo. Data neighboring in local load balancing operations. In *ICCOMP'05: Proceedings of the 9th WSEAS International Conference on Computers*, pages 1–6, Stevens Point, Wisconsin, USA, 2005. World Scientific and Engineering Academy and Society (WSEAS).

[206] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical recipes in C: the art of scientific computing*. Cambridge University Press, New York, NY, USA, 1988.

[207] J. Arabe, A. Beguelin, B. Lowekamp, E. Seligman, M. Starkey, and P. Stephan. Dome: Parallel programming in a distributed computing environment. *Parallel Processing Symposium, International*, 0:218, 1996.

[208] J. Casas, R. Konuru, S. W. Otto, R. Prouty, and J. Walpole. Adaptive load migration systems for pvm. In *Supercomputing '94: Proceedings of the 1994 conference on Supercomputing*, pages 390–399, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[209] F. Douglis and J. Ousterhout. Transparent process migration: design alternatives and the sprite implementation. *Mobility: processes, computers, and agents*, pages 57–86, 1999.

[210] E. T. Roush. Fast dynamic process migration. In *ICDCS '96: Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS '96)*, page 637, Washington, DC, USA, 1996. IEEE Computer Society.

[211] B. S. Siegell. *Automatic generation of parallel programs with dynamic load balancing for a network of workstations*. PhD thesis, May Cmu-Cs- School, Pittsburgh, PA, USA, 1995.

[212] D. L. Eager, E. D. Lazowska, and J. Zahorhan. Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, 12:662–675, 1986.

[213] S. Zhou. An experimental assessment of resource queue length as load indices. In *Technical Report UCB/CSD 86/298*, California, USA, 1986. University of California, Berkeley.

[214] S. Zhou. A trace–driven simulation study of dynamic load balancing. In *Technical Report UCB/CSD 87/305*, California, USA, 1987. University of California, Berkeley.

[215] W. E. Leland and Ott T. J. Load balancing heuristics and process migration. In *Performance '86: Proceedings of the ACM SIGMETRICS Conference*, pages 54–69, Raleigh, North Carolina, USA, 1986.

[216] M. V. Devarakonda. *File Usage Analysis and Resource Usage Prediction: A Meassurement–Based Study*. PhD thesis, University of Illinois at Urbana–Champaign, 1988.

[217] M. V. Devarakonda and K. I. Ravishankar. Predictability of process resource usage: A meassurement–based study on unix. *IEEE Transactions on Software Engineering*, 15(12):1579–1586, 1989.

[218] K. K. Goswami, K. I. Ravishankar, and M. Devarakonda. Load sharing based on task resource prediction. In *22nd Annual Hawaii Info. Conference on System Sciences*, volume 2, pages 921–927, 1999.

[219] K. K. Goswami, M. Devarakonda, and K. I. Ravishankar. Prediction–based dynamic load–sharing heuristics. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):638–648, 1993.

[220] J. Baxter and J. H. Patel. Profiling based task migration. In IEEE Computer Society Press, editor, *6th International Parallel Processing Symposium*, pages 192–197, 1992.

[221] F. Douglis and J. Ousterhout. Transparent process migration: Design alternatives and the sprite implementation. *Software - Practice and Experience*, 21(8):757–786, 1991.

[222] A. Barak, A. Braverman, I. Gilerman, and O. Laden. Performance of pvm with the mosix preemptive process migration scheme. In IEEE Computer Society Press, editor, *7th Israeli Conf. on Computer Systems and Software Engineering*, pages 38–45, 1996.

[223] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor – a hunter of idle workstations. In *8th Intl. Conf. on Distributed Computing Systems*, pages 104–111, 1988.

[224] N. Carriero, E. Freeman, D. Gelernter, and D. Kaminsky. Adaptive parallelism and piranha. *IEEE Computer*, 28(1):40–49, 1995.

[225] R. B. Konuru, S. W. Otto, and J. Walpole. A migratable user–level process package for pvm. *Journal of Parallel and Distributed Computing*, 40(1):81–102, 1997.

[226] L. V. Kale. The chare kernel parallel programming language and system. In *Intl. Conf. on Parallel Processing*, volume 2, pages 17–25, 1990.

[227] R. J. Richards, B. Ramkumar, and S. G. Rathnam. ELMO: Extending (Sequential) Languages with Migratable Objects – Compiler support. In IEEE Computer Society Press, editor, *4th Intl. Conf. on High–Performance Computing*, pages 180–185, 1997.

[228] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In MIT Press, editor, *5th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP 95)*, pages 207–216, 1995.

[229] S. Atlas, S. Banerjee, J. C. Cummings, P. J. Hinker, M. Srikant, J. V-W. Reynders, and M. Tholburn. POOMA : A high–performance distributed simmulation environment for scientific applications. *Supercomputing'95*, pages–, 1995.

[230] J. Arabe, A. Beguelin, B. Lowekamp, E. Seligman, M. Starkey, and P. Stephan. Dome: parallel programming in a heterogeneous multi–user environment. Technical Report CS–95–137, Carnegie Mellon University, School of Computer Sciences, 1995.

[231] M. A. Bhandarkar, R. Brunner, and L. V. Kale. Run-time support for adaptive load balancing. In *IPDPS '00: Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, pages 1152–1159, London, UK, 2000. Springer-Verlag.

[232] J. Roca, J.A. Alvarez, J.J. Fernandez, J.C. Ortega, and I. García. Balanceo dinamico de carga mediante migracion de hebras. In *Actas de las XI Jornadas de Concurrencia. COL.LECCIO TREVALLS DÍNFORMATICA I TECNOLOGIA. N. 16*, pages 67–78, Castellón, 2003.

[233] J.A. Alvarez, J. Roca, J.C. Ortega, J.J. Fernandez, and I. Garcia. Migración de objetos concurrentes para balanceo dinamico de carga. In *XIV Jornadas de Paralelismo*, pages 431–436, 2003.

[234] J.C. Ortega, J.A. Alvarez, and J. Roca. Concurrent object migration preserving data locality. In *Workshop on Charm++ 2003*, Urbana-Champaign, USA, 2003.

[235] J. A. Alvarez, J. Roca, J. C. Ortega, and I. García. Object cluster computing advantages of object oriented load balancing strategies. In *Proceedings of the IADIS International Conference Applied Computing 2004*, pages 147–150, Lisbon, Portugal, 2004.

[236] J.A. Martinez, L.G. Casado, J.A. Alvarez, and I. García. Interval parallel global optimization with charm++. In *Proceedings of the Workshop on State of the Art in Scientific Computing. PARA'04*, page 93–99, Technical University of Denmark, Lyngby, Denmark, 2004.

[237] J.A. Martínez, L.G. Casado, J.A. Alvarez, and I. Garcá. Interval parallel global optimization with charm++. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 3732 LNCS:161–168, 2006.

[238] J.A. Alvarez, J.C. Ortega, and J. Roca. Load balancing strategy using concurrent objects. In I. García, L.G. Casado, V.G. Ruiz, and J.J. Fernandez, editors, *Actas de las XV Jornadas de Paralelismo. Computación de Altas prestaciones.*, pages 243–246, Almeria, Septiembre 2004. Servicio de publicaciones de la Universidad de Almeria.

[239] J. Roca, J.C. Ortega, and J.A. Alvarez. Local load balancing strategy keeping data neighborhood. *WSEAS Transactions on Computers*, 4(8):917–924, 2005.

[240] J. Roca, J.C. Ortega, J.A. Alvarez, and J. Mateo. La localidad de los datos, criterio complementario en operaciones de balanceo dinámico de carga. In *Proceedings of the IADIS International Conference Iberoamericana WWW/Internet 2005*, pages 386–391, Lisbon, Portugal, 2005.

[241] J. Roca, J. C. Ortega, J. A. Alvarez, and J. Mateo. Data neighboring in local load balancing operations. In *ICCOMP'05: Proceedings of the 9th WSEAS International Conference on Computers*, pages 1–6, Stevens Point, Wisconsin, USA, 2005. World Scientific and Engineering Academy and Society (WSEAS).

[242] J.A. Alvarez, J. Roca, and J.J. Fernandez. Concurrent threads for parallel electron tomography. *WSEAS Transactions on Information Sciences and Applications*, 4(4):525–530, 2006.

[243] J.A. Alvarez, J. Roca, and J.J. Fernandez. Multithreaded tomographic reconstruction. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 4757 LNCS:81–88, 2007. cited By (since 1996) 2.

[244] J.A. Alvarez, J. Roca, and J.J. Fernandez. Parallel electron tomography in a multithreaded environment. *WSEAS Transactions on Information Science and Applications*, 4(4):806–813, 2007.

[245] J. Roca, J. A. Alvarez, and J. J. Fernandez. Latency hiding and adaptability for parallel iterative algorithms. In *Proceedings of the Euroepan Computing Conference'07*, 2007.

[246] J. A. Alvarez, J. Roca, and J.J. Fernandez. Exploiting concurrence in 3d parallel iterative reconstruction algorithms. In *Actas de las XVIII Jornadas de Paralelismo*, page 571–578, 2007.

[247] J. A. Alvarez, J. Roca, and J.J. Fernandez. A load balancing framework in multithreaded tomographic reconstruction. In C. Bischof, M. Bücker, P. Gibbon, G.R. Joubert, T. Lippert, B. Mohr, and F. Peters, editors, *Proceedings ParCo 2007 Conference: Parallel Computing: Architectures, Algorithms and Applications*, page 165–172, 2007.

[248] J.A. Alvarez, J. Roca, and J.J. Fernandez. *Parallel Computing: Architectures, Algorithms and Applications*, chapter A Load Balancing Framework in Multithreaded Tomographic Reconstruction, pages 165 – 172. IOS Press, 2008.

[249] J. A. Alvarez and J. Roca. Tomographic image reconstruction using abstractions. *Computer Aided Systems Theory - EUROCAST 2009: 12th International Conference, Las Palmas de Gran Canaria, Spain, February 15-20, 2009, Revised Selected Papers*, pages 334–341, 2009.

[250] J.A. Alvarez and J.R. Piera. High level abstractions for improving parallel image reconstruction algorithms. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5518 LNCS(PART 2):50–57, 2009.

[251] J. Roca, J. A. Alvarez, and J. J. Fernandez. Latency hiding and adaptability for parallel iterative algorithms. In Nikos Mastorakis, Valeri Mladenov, and Vassiliki T. Kontargyri, editors, *Proceedings of the European Computing Conference*, volume 28 of *Lecture Notes in Electrical Engineering*, pages 169–175. Springer US, 2009.

[252] J.A. Alvarez, J. Roca, and J.J. Fernandez. From structured to object oriented programming in parallel algorithms for 3d image reconstruction. In *Proceedings of the 8th Workshop on Parallel/High-Performance Object-Oriented Scientific Computing, POOSC '09*, 2009.

[253] J.A. Alvarez and J. Roca-Piera. A proposed asynchronous object load balancing method for parallel 3d image reconstruction applications. *Algorithms and Architectures for Parallel Processing*, 6081:454–462, 2010.