UNIVERSIDADE DE SANTIAGO DE COMPOSTELA

Centro de Investigación en Tecnoloxías da Información
Departamento de Electrónica e Computación

Tesis doctoral

MODELADO DEL RENDIMIENTO DE CÓDIGOS IRREGULARES PARALELOS EN SISTEMAS DISTRIBUIDOS

Presentada por:
Marcos Boullón Magán

Dirigida por:
Francisco Fernández Rivera
Tomás Fernández Pena
David Miranda Barrós

Septiembre 2015

USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA

Francisco Fernández Rivera , Profesor Catedrático de Universidad del Área de Arquitectura de Computadores de la Universidad de Santiago de Compostela

Tomás Fernández Pena, Profesor Titular de Universidad del Área de Arquitectura de Computadores de la Universidad de Santiago de Compostela

David Miranda Barrós, Profesor Titular de Universidad del Área de Ingeniería Cartográfica, Geodésica y Fotogrametría de la Universidad de Santiago de Compostela

HACEN CONSTAR:

Que la memoria titulada **MODELADO DEL RENDIMIENTO DE CÓDIGOS IRREGULARES PARALELOS EN SISTEMAS DISTRIBUIDOS** ha sido realizada por **Marcos Boullón Magán** bajo nuestra dirección en el Centro Singular de Investigación en Tecnoloxías da Información de la Universidad de Santiago de Compostela, y constituye la Tesis que presenta para optar al título de Doctor.

Septiembre 2015

Francisco Fernández Rivera
Codirector tesis

Tomás Fernández Pena
Codirector de la tesis

David Miranda Barrós
Codirector de la tesis

Marcos Boullón Magán
Autor de la tesis

USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA

In anno MMDCCLXVIII ab urbe condita fecit est.

M.

USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA

Agradecimientos

Quiero expresar mi más sincero agradecimiento a todas las personas que han contribuido a la realización de este trabajo, tanto en el ámbito académico como en el personal. Como éste ha sido un esfuerzo prolongado durante años, son innumerables las personas que entran en esa categoría. Perdonadme si no hago una lista.

En especial quiero dar las gracias a mis codirectores, los profesores D. Francisco Fernández Rivera, D. Tomás Fernández Pena y D. David Miranda Barrós, por toda la ayuda que me han prestado, la paciencia que mostraron conmigo y la confianza depositada en este trabajo. Sería imposible olvidar también el apoyo incondicional del profesor D. Rafael Crecente Maseda, que siempre se tomó este tema con humor y no ha podido ver completado el trabajo. Ha sido una fuente de inspiración desde el primer momento.

Mis agradecimientos a todos los miembros del Departamento de Electrónica e Computación, en especial a los que pertenecen o han pertenecido al Grupo de Arquitectura de Computadores, por haberme hecho compartir momentos maravillosos. Igualmente para todos los miembros presentes y pasados del grupo Laboratorio do Territorio, en el Departamento de Enxeñaría Agroforestal, por estar haciéndolo ahora.

Por último, a mi familia, por ser siempre el mejor soporte.

Este trabajo no hubiera podido completarse sin el soporte económico directo del proyecto CrossGrid, *CrossGrid - Development of Grid Environment for Interactive Applications; Proposal/Contract no. IST-2001-32243*, de otros muchos contratos de proyectos en la USC, y de las infraestructuras de computación del *Centro de Supercomputación de Galicia* (CESGA).

Septiembre 2015

USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA

Índice general

USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA

Introducción

La evolución de las tecnologías de la información nos permite abordar problemas en ciencia e ingeniería cada vez más grandes, complejos y con mejor resolución, obteniendo soluciones adecuadas en un tiempo razonable. Pocas áreas de conocimiento quedan que no se hayan beneficiado de los avances en esta disciplina. La revisión continua de los fundamentos teóricos de la informática, combinado con el desarrollo de metodologías y algoritmos sofisticados para procesar datos y validar hipótesis, junto con la disponibilidad de altas capacidades computacionales, representan las herramientas básicas de nuestra forma de hacer ciencia a día de hoy.

En cualquier caso, la necesidad, o posibilidad, de resolver problemas complejos requieren, cada día más, del uso de grandes infraestructuras computacionales, como son los supercomputadores o la computación Grid. Los grandes centros de supercomputación albergan máquinas cada vez más potentes, con velocidades de procesamiento que superan los petaFLOPs. Por otro lado, la computación Grid permite llevar todavía más allá la disponibilidad de estos recursos, ya que proporciona una mayor simplicidad y transparencia en el uso coordinado de los mismos, ya se trate de recursos puramente computacionales, de almacenamiento y/o aplicaciones específicas, sin necesidad de un control centralizado.

La programación de estas arquitecturas distribuidas es cada vez más popular, y, muchas veces, imprescindible, entre los miembros de la comunidad científica. Hay diferentes formas de tratar con este problema y con frecuencia surgen nuevos desarrollos que se centran en alguno de sus puntos fuertes. En este escenario, factores como la heterogeneidad de los nodos de computación y las redes de interconexión o la localidad de los datos resultan ser de la mayor importancia para optimizar el uso de los recursos. Redundancia, resistencia y escalabilidad son otros valores a tener en cuenta.

Muchas veces se deben reciclar viejas técnicas para tratar con las nuevas arquitecturas. Así, los métodos de predicción de rendimiento, usados frecuentemente tanto por los arquitectos de computadores para evaluar el diseño de nuevos sistemas, como por los creadores de compiladores para explorar nuevas optimizaciones o los desarrolladores avanzados para afinar sus códigos, pueden ser adaptados para tener en cuenta las peculiaridades de los sistemas distribuidos. Poder conocer a priori características de la ejecución de un algoritmo en cada nodo del sistema, según los recursos disponibles en el mismo y la carga de trabajo presente y futura, ayuda a planificar la distribución de trabajos a lo largo de todo el sistema computacional. Pero no todos los algoritmos pueden ser tratados de la misma forma. Hay una categoría especial de problemas, llamados códigos irregulares, en los que cálculos, control de flujo, patrón de acceso a memoria y comunicaciones durante la ejecución dependen fuertemente de los datos de entrada, y no pueden ser caracterizados correctamente de forma estática. Estos problemas aparecen con frecuencia en el software científico y de ingeniería.

No hay una clasificación estándar de códigos irregulares, pero hay algunas situaciones básicas. Una aplicación irregular accede a posiciones de memoria no estructuradas que varían durante la ejecución. O contiene niveles de indireccionamientos que impiden determinar, en tiempo de compilación, el conjunto de accesos a memoria realizados por la aplicación. Durante este trabajo se considera un tercer tipo, una aplicación irregular donde el control de flujo, accesos a memoria y operaciones de cálculo durante la ejecución dependen fuertemente de los datos de entrada y de la evolución del estado, y que no pueden ser caracterizados con precisión de manera estática. Los códigos irregulares pueden estar localizados en las secciones críticas del programa y se caracterizan por tener un mal comportamiento en la evaluación de rendimiento. Presentan gran dificultad de análisis estáticos, muestran una explotación compleja e ineficiente de la jerarquía de memoria, implementan estructuras de datos complejas con las cuales se logra un aumento del rendimiento del programa mediante una reducción del volumen de datos almacenados, y alternan entre vías de ejecución del código alternativas en un patrón que no puede ser caracterizado a priori.

En este documento describimos una metodología que permite crear modelos de predicción de rendimiento de códigos irregulares para arquitecturas paralelas y distribuidas, metodología que se ha aplicado a un grupo escogido de algoritmos sobre diferentes tipos de sistemas. Los algoritmos seleccionados corresponden todos a aplicaciones reales, que representan distintas categorías de códigos irregulares. Estos algoritmos cubren diferentes ámbitos de interés en

ciencia y tecnología, incluyendo desde la resolución de sistemas de ecuaciones con matrices dispersas hasta métodos heurísticos de optimización. Sobre todos estos algoritmos se han llevado a cabo diferentes tipos de análisis para caracterizar su comportamiento ante diversas configuraciones de los sistemas computacionales y de los datos de entrada. Este proceso supone distintas aproximaciones al problema de la modelización de códigos irregulares.

Objetivos

El objetivo principal de este trabajo es explorar y desarrollar distintas aproximaciones metodológicas a la modelización analítica de códigos irregulares paralelos. Para ello nos centramos en los siguientes subobjetivos:

1. Obtener un modelo temporal para comunicaciones en el entorno de desarrollo del proyecto CrossGrid
2. Obtener un modelo de computaciones y comunicaciones para los métodos iterativos implementados en la librería PARAISO
3. Obtener un modelo de computaciones y comunicaciones para la versión paralela del simulador de dispersión de contaminantes STEM-II
4. Obtener un modelo de computaciones para un algoritmo de optimización estocástica basado en el *Simulated Annealing* de la herramienta de planificación territorial RULES. Paralelizarlo, y obtener a continuación el modelo de comunicaciones

Metodología

El presente trabajo está centrado en códigos paralelos irregulares, donde el control de flujo, accesos y cálculo durante la ejecución dependen fuertemente de los datos de entrada, y no pueden ser caracterizados con precisión de manera estática. Los valores de los parámetros que rigen el comportamiento son establecidos estadísticamente a partir del análisis del algoritmo ante distintas combinaciones en los parámetros de entrada.

Con el objetivo de obtener modelos analíticos para estos códigos, hemos establecido una metodología basada en mediciones exhaustivas de tiempos de ejecución en un entorno controlado

o, alternativamente, mediciones del número de operaciones en punto flotante, comunicaciones, etc. La idea principal es correlacionar las resultados de rendimiento y los valores de la magnitudes monitorizadas, lo que ha permitido establecer expresiones analíticas del rendimiento que permitan predecirlo con precisión en diferentes escenarios.

La metodología se divide en dos etapas. En la primera se estudia del comportamiento de los códigos en una ejecución estándar, que servirá para determinar los parámetros más influyentes en el modelo final. En la segunda etapa se obtienen los modelos correlacionando las mediciones bajo distintas configuraciones del sistema y de los datos de entrada, utilizando un rango adecuado de valores en los parámetros.

Inicialmente los códigos son caracterizados estáticamente en términos del número preciso de eventos relevantes, por ejemplo, el tiempo de ejecución o el número de FLOPs requeridos por el código. Este número es contabilizado manualmente a partir del conocimiento del algoritmo, analizando el código fuente, o automáticamente, mediante el uso de los contadores hardware disponibles en los procesadores del sistema. Este resultado es finalmente expresado con ecuaciones algebraicas que involucran parámetros del código y de los datos de entrada, como son, entre otros, el tamaño de las matrices de entrada, el número de elementos no nulos que contienen, el grado polinomial del precondicionador de un resolutor iterativo, el número de iteraciones en un lazo crítico o diferentes parámetros de control del usuario.

En resumen, a partir de información estática básica sobre computaciones y comunicaciones, hemos desarrollado modelos de rendimiento de los diferentes códigos. Utilizando herramientas de monitorización, el coste de cada código se evalúa a través de un número significativo de ejecuciones bajo diferentes condiciones iniciales para obtener los modelos analíticos finales de computación y comunicaciones. La idea de la metodología es obtener una correlación entre la información estática y de monitorización y las características intrínsecas del código con los tiempos de ejecución o el número de operaciones en punto flotante medidos, por ejemplo.

Organización del trabajo

Capítulo 1

En este capítulo se hace una breve revisión de aspectos esenciales de la computación paralela necesarios para el trabajo. Primeramente, se discute la clasificación tradicional de estos sistemas, con un interés especial en el modelo de computación Grid. A continuación, se muestran

los paradigmas de programación paralela, para finalizar centrándose en el análisis de rendimiento de las arquitecturas paralelas: metodologías desarrolladas para evaluar el rendimiento, métricas para simplificar en unos pocos valores toda la información del sistema, límites de aplicabilidad, modelos analíticos paralelos tradicionales y herramientas software de apoyo en la tarea.

Capítulo 2

Este capítulo examina la implementación de una librería de pase de mensajes en el Grid. Se inicia con una descripción breve de los principios básicos de este paradigma de programación paralela, en un sistema basado en la plataforma Globus, para, a continuación, pasar a construir un modelo que caracteriza el tiempo de ejecución de las operaciones de comunicación punto a punto y colectivas. El análisis y modelado posterior están diseñados para el software utilizado en el proyecto CrossGrid, en desarrollo entre 2002 y 2005. En la actualidad, existen nuevas versiones del software Grid para las que es posible que el análisis llevado a cabo no sea totalmente aplicable a los sistemas actuales, pero, en cualquier caso, los fundamentos de la metodología desarrollada son válidos.

Capítulo 3

Este capítulo analiza y modeliza una librería de álgebra matricial dispersa que hace uso simultáneamente de paralelismo de datos y paralelismo por pase de mensajes. En primer lugar, se muestran someramente las principales técnicas de resolución iterativa de sistemas lineales de ecuaciones. A continuación, se discuten las características de la librería PARAISO. Finalmente, se caracterizan las computaciones y comunicaciones de la librería para definir un modelo de coste de los diferentes algoritmos que la componen.

Capítulo 4

Este capítulo examina un programa de simulación de contaminación atmosférica basado en modelos meteorológicos, centrándose en la rutina que consume la mayor parte de su tiempo de ejecución. Se presenta brevemente el problema al que está orientada la simulación, para continuar con un análisis del código que ha permitido identificar la parte más costosa. Una vez determinada la rutina a estudiar, se ha llevado a cabo el modelado de sus computaciones y comunicaciones.

Capítulo 5

Este capítulo tiene como objetivo paralelizar y modelizar el rendimiento de un algoritmo estocástico de optimización usado en una herramienta de ordenación del territorio para la formulación de planes de uso del suelo. Para comenzar, se muestran brevemente los fundamentos teóricos y características particulares de esta implementación, para continuar con la identificación, análisis y modelado de las secciones útiles con las que caracterizaremos computaciones y comunicaciones. En este caso, el algoritmo de partida es secuencial, por lo que se propone y modela una paralelización en MPI a sistemas multiprocesador.

Conclusiones

En este capítulo se resumen las principales aportaciones del trabajo y se enumeran las principales publicaciones derivadas del mismo.

Capítulo 1

Modelos de rendimiento en sistemas paralelos

En este capítulo se hace una breve revisión de aspectos esenciales de la computación paralela necesarios para el trabajo. Primeramente, se discute la clasificación tradicional de estos sistemas, con un interés especial en el modelo de computación Grid. A continuación, se muestran los paradigmas de programación paralela, para finalizar centrándose en el análisis de rendimiento de las arquitecturas paralelas: metodologías desarrolladas para evaluar el rendimiento, métricas para simplificar en unos pocos valores toda la información del sistema, límites de aplicabilidad, modelos analíticos paralelos tradicionales y herramientas software de apoyo en la tarea.

1.1. Arquitecturas paralelas

Un sistema paralelo es una colección de elementos de procesamiento independientes que cooperan y se comunican entre sí con la finalidad de resolver un problema de forma más eficiente de la que resultaría al utilizar sólo uno de ellos [215]. Es una definición poco precisa que abarca distintos niveles de paralelismo y puede referirse tanto a los sistemas multinúcleo como a la computación Grid y Cloud, por ejemplo.

Diferentes aproximaciones al paralelismo dificultan una única clasificación de las arquitecturas paralelas. Una clasificación es la de Flynn [94], que se basa en el movimiento de instrucciones y datos por el conjunto del sistema. Diferencia entre SISD (*Simple Instruction, Simple*

Data), MISD (*Multiple Instruction, Simple Data*), SIMD (*Simple Instruction, Multiple Data*) y MIMD (*Multiple Instruction, Multiple Data*).

Una clasificación más útil de los sistemas paralelos utilizados en entornos de computación de altas prestaciones (HPC, *High Performance Computing*) se basa en el paradigma de programación. Básicamente distingue entre tres grandes familias: procesamiento paralelo de datos o *stream processing*; arquitecturas de memoria compartida, llamadas multiprocesadores; y arquitecturas distribuidas, llamadas multicomputadores. Esta clasificación es genérica y existen aproximaciones diferentes, pero es una aproximación suficiente para los sistemas más habituales.

1.1.1. Procesamiento paralelo de datos

En estos sistemas una operación se ejecuta en paralelo en cada elemento de una estructura regular, actuando una misma instrucción sobre diferentes datos. Este modelo encaja en la categoría de SIMD y es la base de los procesadores vectoriales. Las mismas técnicas siguen vigentes en otras áreas como la programación de GPUs. Los sistemas híbridos CPU y GPU obtienen buenos rendimientos en muchas aplicaciones [159][209][231][62].

1.1.2. Memoria compartida

Los multiprocesadores intercambian información a través de un espacio de memoria compartida. Todos los elementos de computación pueden leer cualquier dirección de la memoria del sistema utilizando las instrucciones básicas de acceso a memoria. El mecanismo más popular para programar estos sistemas es OpenMP [3] que, mediante directivas de compilación, proporciona una interfaz sencilla para generar automáticamente código paralelo.

Los multiprocesadores, de acuerdo a cómo se implemente la memoria compartida [58], pueden ser procesadores simétricos, llamados SMP (*Symmetric Multiprocessors*) o UMA (*Uniform Memory Access*), o sistemas NUMA (*Non Uniform Memory Access*) [126][67][210][60].

Los UMA (figura 1.1) son sistemas fuertemente acoplados donde los accesos a memoria tienen un retardo independiente de su localización, aunque pueden incorporar cache privadas que matizan esta característica. Utilizan un bus de comunicaciones para conectar procesador y memoria, garantizando un mismo coste de acceso a memoria independientemente de la dirección solicitada. Los problemas de contención en el buffer limitan la escalabilidad. Se

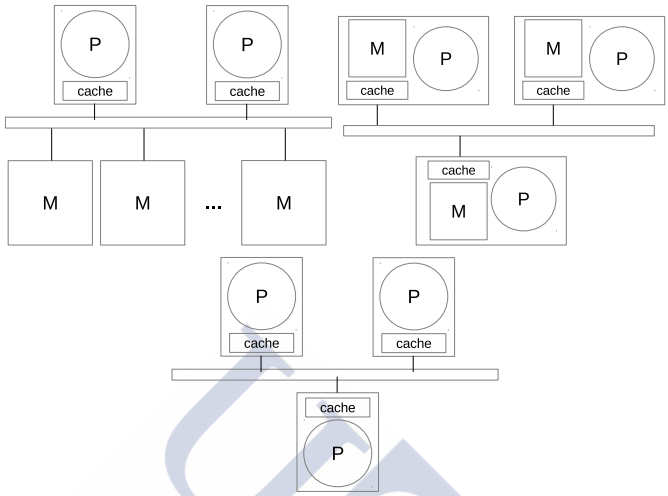

Figura 1.1: Arquitecturas de memoria compartida. (1) UMA. Cada nodo está formado por un procesador y una cache. La memoria es global. (2) NUMA. Cada nodo está formado por un procesador, su propia memoria local y una cache. (3) COMA. Cada nodo está formado por un procesador y una cache. Los datos están repartidos entre los nodos y cada solicitud de lectura/escritura los desplaza.

diferencia entre UMA simétrico, donde todos los procesadores pueden ejecutar el software de sistema, y UMA asimétrico o con procesadores dedicados, donde hay procesadores de sistema y procesadores de cálculo [182][149].

En los sistemas NUMA la memoria está físicamente distribuida por los nodos del sistema, pero existe una memoria lógica compartida gracias a la red de interconexión. Esto provoca costes diferentes según la distancia entre el procesador solicitante y la dirección solicitada, donde pueden existir caches privadas, y esta jerarquía de memoria produce latencias de acceso heterogéneas [138][155]. En esta versión, los procesadores tienen acceso privilegiado de baja latencia a un nivel la memoria, que es local a ellos, mientras que localizaciones de memoria asociadas a otro procesador implican mayor latencia, que depende de la jerarquía de memoria implementada en el sistema. Reduciendo la memoria compartida de un sistema NUMA a sólo memorias caches privadas, obtenemos arquitecturas COMA (*Cache-Only Memory*, con un único nivel de jerarquía de memoria y directorios cache distribuidos por todo el sistema [211][221].

1.1.3. Arquitecturas distribuidas

Los multicomputadores están formados por una colección de elementos computacionales (nodos) independientes y con procesador, memoria y entrada y salida individualizadas, conectados por una red de interconexión [65]. Un nodo no puede acceder a la memoria de otro. Para el acceso a los datos no locales se exige una comunicación explícita entre los implicados. Los intercambios de datos y sincronizaciones utilizan un protocolo de pase de mensajes.

El paradigma de programación por pase de mensajes es el mecanismo de programación habitual en estos sistemas, con MPI [97][159] como protocolo principal. Son altamente escalables, flexibles y versátiles, y las arquitecturas más comunes para grandes supercomputadores en los últimos años.

Las arquitecturas distribuidas se clasifican según el nivel de acoplamiento entre unidades de procesamiento en clusters, constelaciones y computadores masivamente paralelos [10].

Un cluster es, en esencia, un sistema formado por procesadores independientes conectados a través de una red dedicada [30][80][52][183]. Son sistemas poco acoplados, robustos, flexibles. La latencia y ancho de banda típicos de estos sistemas puede variar en órdenes de magnitud frente a las empleadas en supercomputadores. Al mismo tiempo, el sistema operativo y software de operación de estos sistemas es genérico en lugar de estar diseñado y optimizado en exclusiva para la arquitectura como en los multicomputadores acoplados. Aunque estas diferencias implican un rendimiento subóptimo respecto a arquitecturas alternativas, su popularidad proviene de una buena relación precio/rendimiento, ya que la mayor parte de estos sistemas están construidos con componentes genéricos para nodos y para la red. En relación a otros supercomputadores, los clusters ofrecen una mayor flexibilidad a la hora de integrar los componentes del sistema en función de las necesidades del usuario final, sin incurrir en gastos adicionales para una configuración a medida. Permiten una rápida respuesta a los avances tecnológicos, incorporando con facilidad nuevos dispositivos [200].

La configuración más sencilla del cluster consiste en PCs comunicados por una variante de red Ethernet. Actualmente la mayoría de procesadores en PCs son multinúcleo, que produce niveles de comunicación internodo e intranodo. Los programas que mejor se adapten a esta jerarquía de comunicaciones serán más eficientes.

Las constelaciones es un subtipo de cluster, donde el número de procesadores por nodo sobrepasa al número de nodos del sistema, y por tanto el paralelismo más importante para explotar

es el intranodo [80]. En general, sus nodos son sistemas de memoria compartida UMA o NUMA, y el paradigma de programación es un híbrido entre pase de mensajes entre nodos y paralelismo de datos dentro del nodo.

Por último, las arquitecturas masivamente paralelas son sistemas distribuidos con un nivel de paralelismo muy alto. En los computadores masivamente paralelos, MPP (*Massively Parallel Processors*), los nodos son independientes, con un nivel de acoplamiento muy alto, utilizando una red de interconexión de alta velocidad para gestionar el conjunto como un único sistema multiprocesador.

Puesto que la arquitectura está optimizada y el hardware diseñado específicamente para estas arquitecturas, obtienen muy buen rendimiento a costa de precios altos.

1.1.4. Arquitecturas altamente distribuidas

Dos tipos de arquitecturas distribuidas se han desarrollado y popularizado en la última década de manera que requieren una mención especial: la computación Grid [8] y la computación Cloud [21].

La computación Grid es una evolución de las arquitecturas de memoria distribuida, equivalente a un cluster distribuido geográficamente, heterogéneo y al cargo de dominios administrativos diferentes.

Es una arquitectura distribuida optimizada para acceder a recursos no locales y pertenecientes a dominios administrativos diferentes. A pesar de que los recursos no están gestionados por un único control centralizado, pueden ser coordinados para ofrecer capacidades de computación agregadas. De esta forma puede coordinar recursos heterogéneos en gran número, pertenecientes a distintas instituciones, y hacer que cooperen en un objetivo común. Necesita de la gestión dinámica de recursos, pero también de una infraestructura de seguridad. El objetivo es procesar grandes problemas de la ciencia y la industria sin necesidad de depender de sistemas específicos.

Por otra parte, la computación Cloud es un nuevo modelo de prestación de servicios de computación o de negocio a través de la virtualización de recursos, que obliga a que por debajo se implemente un sistema distribuido, y que permite al usuario acceder a un catálogo de servicios estandarizados y trabajar en un entorno que se adapte dinámicamente a sus necesidades

técnicas [21]. Se puede entender como una reserva de recursos hardware, plataforma de desarrollo o servicios, que pueden ser asignados dinámicamente al usuario que los solicite según la evolución de su carga de trabajo [90][189].

1.1.5. Redes de interconexión

La escala a la que se implementan las arquitecturas paralelas actuales convierte a la red de comunicación en un elemento clave de la eficiencia del sistema. La topología y características técnicas tienen un impacto crucial en la ejecución de códigos paralelos.

Las redes de interconexión se clasifican en cuatro tipos básicos que, a su vez, se dividen en subclases [81][174]: redes de medio compartido, redes directas o estáticas, redes indirectas o dinámicas y redes híbridas. En una red directa, los enlaces punto a punto interconectan los nodos el sistema multicomputador de forma fija. Las topologías regulares como mallas, toroides o hipercubos eran habituales en los centros de cálculo, mientras que ahora se prefieren mallas simples. En el caso de redes indirectas, se permite una variación en la conexión de los nodos de procesamiento, lo que implica alguna forma de sistema de conmutación. Aquí se encuentran ejemplos de *fat trees* o redes multietapa.

Las familias de redes de comunicación que tienen una presencia importante en los centros de cálculo son las extensiones del estándar Ethernet (Gigabit Ethernet, 10-Gigabit Ethernet), y las de la familia InfiniBand, que fue diseñada para ser escalable y proporciona baja latencia, alto ancho de banda, tolerancia a fallos y soporte para QoS.

La primera de ellas realiza comunicaciones a través de funciones del kernel. Estas comunicaciones requieren que la memoria sea copiada ente buffers de usuario y del kernel, que ejecuta la comunicación y lleva los datos al destino, que los vuelca a memoria de usuario. En esta tecnología, el exceso de copia de datos añade latencia a la propia red. Además, por cada mensaje se procesa la pila TCP/IP.

Para mejorar las latencias [130], la segunda familia utiliza protocolos que le permiten comunicarse directamente a través de la capa de usuario sin la intervención del kernel ni interpretar TCP/IP. los datos se mueven directamente de memoria de usuario origen a memoria de usuario destino. Estos protocolos pueden ser reutilizados para implementar las capas de TCP y UDP y que, de esta forma, las comunicaciones del kernel puedan beneficiarse.

Otras alternativas presentes en los centros de cálculo son redes Cray o Myrinet, redes asíncronas y con control distribuido, y las interconexiones propietarias y hechas a medida.

1.2. Tecnologías Grid

El Grid es un paradigma de computación distribuida que hoy día trata de responder a dos necesidades: disponer de entornos para resolver grandes problemas con altas demandas de capacidad computacional [25][77], y simplificar la cooperación entre grupos interdisciplinares geográficamente dispersos [104][100][103].

Aunque distintos centros de cálculo ofrecían sistemas con arquitecturas distribuidas en los años 90, la necesidad de mayor cálculo relanzó la investigación en este campo. En una primera etapa los desarrolladores se concentraron en diseñar nuevas formas de combinar recursos existentes en nuevas aproximaciones al problema. No había ningún tipo de estándar, era una exploración de posibilidades tecnológicas, y el énfasis estaba en el rendimiento con sistemas ad hoc antes que en la cooperación o extensibilidad de las soluciones.

El primer Grid moderno aparece en el proyecto I-WAY [31][11], del año 1995, donde se presentó un sistema que unía múltiples centros de cálculo de EEUU con una red de alta velocidad. Se creó un superordenador virtual y se ejecutaron aplicaciones sobre él en una demostración de sus posibilidades. Ya desde el principio trabajó los aspectos de la seguridad y del envío y gestión de trabajos.

I-WAY es también el germen de The Globus Toolkit [98][103], uno de los *middleware* Grid más utilizados y que permitió empezar una estandarización de las plataformas Grid sobre una base común. Aparecieron otros middleware, como Legion [119] o UNICORE [193][18], pero el Globus Toolkit se estableció como la plataforma Grid de referencia. Incluye componentes software para desplegar de forma sencilla la infraestructura Grid y proporcionar servicios con los que gestionar los aspectos de seguridad, ejecución de trabajos y gestión de recursos. Esta plataforma permite que los recursos trabajen de forma coordinada y proporcionen un acceso a ellos transparente a los usuarios. Versiones posteriores de Globus Toolkit se enfocaron en la interoperabilidad entre Grids, no solo a nivel de desarrollo de aplicaciones que pudieran migrarse entre Grids, sino al nivel de desarrollo de componentes fundamentales que convivieran dentro de un mismo Grid.

En 2002 se funda la OGSA [99], que define un estándar para los componentes fundamentales de seguridad, gestión de recursos, administración de trabajos, interoperabilidad entre usuarios... y que busca una convergencia entre tecnologías Grid y servicios web. Pretende establecer una arquitectura abierta basada en servicios para desarrollar entornos Grid a través de implementar servicios con estado.

En 2003 surge la OGSI [223], que detalla una implementación de la arquitectura propuesta por OGSA mediante una versión modificada de los servicios web, servicios Grid, que permite incorporar estado a los nuevos servicios (GWSDL [2]). La versión 3 de The Globus Toolkit toma esta aproximación.

En 2004 surge una nueva implementación de la arquitectura OGSA que no dependía de una diferenciación entre servicios web sin estado y servicios Grid con estado. Esta especificación es WSRF [217] e integra OGSA con servicios web puros, aunque sigue usando el término servicio Grid para referirse a un servicio web que proporcione una interfaz WSRF. La versión 4 de The Globus Toolkit se basa en esta implementación.

En respuesta a esta nueva tendencia la Unión Europea tomó un papel activo en fomentar y subvencionar las actividades orientadas a Grid dentro del *Quinto Programa Marco* (FP5), principalmente mediante el programa *Information Society Technology* (IST) que dirigía las iniciativas de computación relacionadas con el concepto de Sociedad de la Información. Los proyectos europeos de Grid se organizaron cronológicamente en tres grandes grupos, con supuestos y objetivos diferentes [176][50] (ver figura 1.2).

Proyectos que comenzaron en 2001

Hay tres proyectos en esta categoría: EuroGrid, DataGrid y DAMIEN.

EuroGrid es un proyecto que sirve como demostración del uso de Grid paneuropeos en comunidades científicas e industriales importantes, y se enfoca en los requisitos específicos de un Grid orientado a ellas. Establece centros de trabajo líderes en tecnología Grid, da soporte al *middleware* UNICORE, desarrolla nuevos componentes de Grid (*resource brokers* dinámicos, sistemas de contabilidad, interfaces para acoplamiento de aplicaciones y acceso interactivo) y sirve como plataforma de evaluación de software Grid orientado a áreas significativas: simulaciones biomoleculares, predicción meteorológica, CAE y computación HPC.

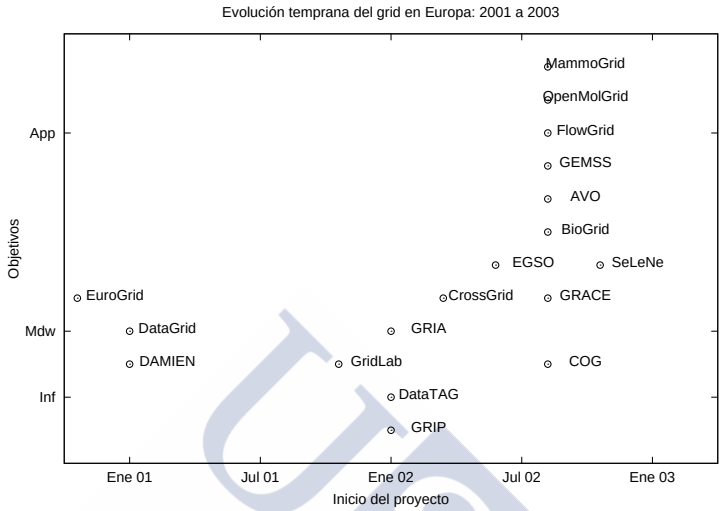

Figura 1.2: Primeros proyectos europeos de computación Grid, orientados a infraestructura, *middleware* y aplicaciones de usuario.

DataGrid [116] tiene como objetivo permitir el acceso a capacidad computacional y almacenamiento distribuidas geográficamente y pertenecientes a organizaciones diferentes. Su orientación es construir middleware para colaboración. Esto proporcionará los recursos necesarios para procesar altos volúmenes de datos provenientes de experimentos de tres disciplinas: física de altas energías, biología y observación de la tierra.

DAMIEN es un proyecto que continúa en la misma línea del proyecto europeo METODIS, orientado a la industria. Su objetivo es continuar desarrollando bloques de construcción para un middleware que permita simulación industrial distribuida y visualización en el Grid. Se centra en facilitar la computación paralela mediante pase de mensajes, acoplamiento de aplicaciones, análisis de rendimiento en aplicaciones finales y visualización.

Proyectos que comenzaron a principios de 2002

Aparecen otros siete proyectos: AVO, GridLab, CrossGrid [49], EGSO, GRIA, DataTAG y GRIP. La mayoría de éstos se desarrollan sobre los productos de los anteriores (fundamentalmente middleware de DataGrid). Siguen orientados a diseñar infraestructura básica a través de la optimización de aplicaciones específicas, y a extender su funcionalidad para cubrir nuevas

áreas de interés: fusión de datos dispersos y herramientas en una única organización virtual, entornos Grid altamente dinámicos, ejecución interactiva de aplicaciones, fusión de fuentes de datos heterogéneas en un único repositorio virtual, desarrollo de modelos de negocio para un mercado de recursos computacionales entre ellos. El proyecto DataTAG trata de interconectar sistemas de ejecución europeos (formato DataGrid) y de socios americanos (formatos iVDGL, GriPhyN), y el proyecto GRIP trata de introducir compatibilidad entre Globus y UNICORE.

Proyectos que comenzaron a finales de 2002

Nueve proyectos pertenecen a esta categoría: FlowGrid (simulaciones de fluidos), OpenMolGrid (entorno para resolución de problemas de diseño molecular), GRACE (búsqueda y catalogación descentralizada de texto no estructurado), COG (unificación de datos heterogéneos con metodologías ontológicas), MOSES (desarrollos en web semántica), BioGrid (interacción de software, metodologías y datos para la industria biotecnológica), GEMMS (simulación avanzada para la planificación quirúrgica preoperatoria), SeLeNe (web semántica para recursos educativos) y MammoGrid (gestión y compartición de imágenes médicas). La mayoría de éstos están dedicados al área de aplicaciones específicas y no a aspectos de middleware o servicios, ya que generalmente reutilizan soluciones ya desarrolladas por proyectos anteriores.

Entre todo este conjunto de proyectos se observa que la tendencia en la investigación en Grid se mueve desde el desarrollo de middleware al desarrollo de aplicaciones. El esfuerzo de los proyectos seminales como EuroGrid y DataGrid en el desarrollo de estándares comunes, y evaluación de paquetes y componentes, permitieron que los últimos esfuerzos construyeran sobre lo que estaba disponible y extendieran la funcionalidad del Grid con nuevos problemas: las nuevas iniciativas estaban fundamentalmente orientadas a aplicaciones, aún sin ignorar del todo las capas de middleware y servicios. Además, las iniciativas de computación evolucionaron desde iniciativas a largo plazo a proyectos más pequeños y enfocados a un único problema. Y las aplicaciones dejan de ser en exclusiva de computación intensiva, y van apareciendo aplicaciones de alto nivel para indexación, catalogación y acceso a recursos y datos heterogéneos.

Además de estas proyectos se encuentran iniciativas en Grid a nivel nacional (NorduGrid, GRIDPP, AstroGrid, GEODISE, GGSA-DAI...), e internacionales (GrADDS, GriPhyN, IPG, iVDGL, PPDG, TeraGrid...).

Como las directrices del *Sexto Programa Marco* (FP6) de la Comisión Europea incluían una mejor coordinación de las actividades de investigación europeas y objetivos, surge la necesidad de consolidar las aproximaciones técnicas de proyectos de Grid en Europa. Para cumplir este objetivo, los principales proyectos europeos fueron agrupados en la GridStart Initiative, lanzada en 2002.

De manera similar, el consorcio EGEE [1] fue establecido para crear una infraestructura Grid de calidad en producción sobre la infraestructura existente de centros de investigación nacionales, reaprovechando los esfuerzos de los proyectos europeos pioneros y en colaboración con la industria. Actualmente ha sido reemplazado por la organización EGI (*European Grid Infrastructure*).

1.2.1. Arquitectura básica de un Grid

The Globus Toolkit 2.4 [98][103] es la implementación de referencia del Grid que hemos utilizado en este trabajo. Se estructura en capas [105], de manera que las capas más próximas a servicios y aplicaciones de usuario pueden utilizar protocolos e interfaces que les abstraen de los desarrollos y tecnologías de las capas inferiores, que gestionan el uso de recursos locales. Incluye un conjunto de protocolos, servicios y herramientas que facilitan la implementación de un sistema Grid, y el desarrollo de aplicaciones que puedan aprovechar el sistema.

En esta plataforma distinguimos:

- Una capa de infraestructura [104], que proporciona el acceso de bajo nivel sobre recursos para explotar en las otras capas.
- Una capa de conectividad, con protocolos básicos para autenticación y comunicación segura.
- Una capa de recursos, con protocolos necesarios para gestionar el acceso e interacción con los recursos y servicios.
- Una capa colectiva, con protocolos para gestión de recursos múltiples: diagnóstico, monitorización, etc.
- Una capa de aplicaciones, formada por las aplicaciones de usuario para dar respuesta a un problema único.

Esta versión se estructura en tres bloques principales: gestión de recursos, de datos y servicios de información. Sobre ellos se construyen las herramientas y aplicaciones de usuario. Y todo ello descansa sobre una infraestructura de seguridad que lo controla todo, para garantizar comunicaciones seguras entre los elementos del Grid, y autenticación y autorización de usuarios entre fronteras administrativas.

El componente básico para la seguridad es el *Grid Security Infrastructure* (GSI), que permite la identificación única del usuario ante todos los recursos del Grid a través de criptografía de clave pública (PKI) y certificados X.509. Satisface la necesidad de ofrecer acceso simplificado a los usuarios, y delegar sus credenciales a los trabajos que se ejecuten en cualquier parte del Grid: el usuario se autentica una única vez, localmente, y ya puede utilizar los recursos para los que esté autorizado.

El componente para la gestión de recursos es el *Grid Resource Allocation Manager* (GRAM). Permite la ejecución remota de un trabajo interpretando las peticiones de los usuarios y ofreciendo los comandos que serán ejecutados en los recursos reservados. Las peticiones se expresan en un lenguaje (RSL) que permite describir de forma estructurada las características del trabajo: necesidad de recursos, ficheros de datos, ejecutables, argumentos, límites, prioridades, etc.

El componente de gestión de datos es el *Global Access to Secondary Storage* (GASS). Gestiona globalmente los ficheros de datos almacenados a lo largo de todo el Grid. Se encarga de transferir los ficheros requeridos por el usuario a los recursos que se ha asignado para la ejecución del trabajo. Para esta tarea se apoya en otros componentes presentes en el Toolkit: GridFTP, una herramienta y protocolo optimizados para transferencias entre recursos físicamente distribuidos; *Globus Replica Catalog*, un catálogo de las distintas direcciones físicas que tiene cada fichero replicado en el Grid; y *Globus Replica Manager*, que decide qué copia del fichero va a ser transferida.

El componente para información es el *Monitoring and Discovery Service* (MDS). Define una infraestructura de directorios uniformes para todos los recursos basada en el estándar para servicios de directorio X.500 y LDAP [131]. Se apoya en el *Grid Resource Information Service* (GRIS), que recoge las características de los recursos localmente, y en el *Grid Index Information Service* (GIIS), que coordina los servicios para ofrecer una visión global del sistema.

Se han desarrollado entornos que facilitan el acceso y uso de estos sistemas a usuarios no especializados, para ignorar los detalles de implementación, pasando de la línea de comando a implementar interfaces web. Es una evolución importante porque permite al usuario acceder al Grid en cualquier momento y lugar a través de una interfaz conocida y que no requiere instalación de componentes específicos.

1.2.2. El proyecto CrossGrid

CrossGrid [49][48][5] fue un proyecto europeo de exploración de las posibilidades del concepto de computación interactiva en Grid. Mientras otras instalaciones Grid sólo permiten que las aplicaciones se ejecuten en lotes y utilizan el resultado final para tomar decisiones de control, con CrossGrid era posible monitorizar el progreso de la ejecución de la aplicación, cambiar sus parámetros y recibir información de salida al momento. Extendía la funcionalidad del Grid cubriendo un nueva categoría de aplicaciones, aquellas que requerían interacción frecuente con un agente humano y, por tanto, tiempos de respuesta rápidos [48][5].

El objetivo básico del proyecto era maximizar el rendimiento de un conjunto de aplicaciones piloto seleccionadas entre distintas áreas de la ciencia y, al mismo tiempo, identificar y resolver los problemas que surjan en el Grid interactivo. Por supuesto, el sistema final todavía era capaz de ejecutar aplicaciones genéricas.

Las características más innovadoras del proyecto CrossGrid fueron orientadas a desarrollar aplicaciones interactivas en el Grid. Pero también promocionó otros componentes para ser reutilizados en nuevos proyectos:

- El Migrating Desktop/RAS [150][148]: un método orientado al usuario para acceder a los recursos del Grid desde cualquier ordenador conectado a la web, con entornos personalizados para cada usuario reconocido por el servicio.
- El planificador de tareas CrossGrid, que soporta colas de trabajos paralelos y secuenciales para múltiples usuarios, interactivos o desatendidos por lotes, expresados en un lenguaje de descripción de trabajos que reconoce prioridades, y con mecanismos de supervisión.
- Las herramientas OCM-G/G-PM, para monitorización avanzada del rendimiento de aplicaciones durante su ejecución.

La tarea del Departamento de Electrónica e Computación de la USC dentro del proyecto CrossGrid consistió en construir el *Performance Predictor Component* (PPC), una herramienta que proporcionaba información sobre el comportamiento de códigos críticos durante su ejecución en el Grid. Contenía el modelo analítico de rendimiento para un grupo escogido de algoritmos, que representaban distintas categorías de códigos irregulares. El cálculo de estos modelos implicó distintas aproximaciones al problema de la modelización de códigos irregulares paralelos que se muestran en este trabajo. Los códigos considerados eran de bajo nivel, útiles para cualquier usuario que utilizara MPI en CrossGrid, y de alto nivel, computacionalmente intensivos y dependientes de la aplicación específica.

El PPC incluía una herramienta de visualización de la información de rendimiento integrada en el Migrating Desktop. Los resultados mostrados para cada modelo dependían de:

- El estado del Grid. PPC se conectaba con los servicios de monitorización y de predicción del estado para conocer la ocupación y disponibilidad de recursos en el sistema, o se definía de manera manual un estado deseable. La información necesaria dependía de la aplicación concreta, pero los parámetros básicos eran la potencia computacional del nodo, y el ancho de banda y latencia entre cada pareja de nodos.
- El código seleccionado. Estos modelos necesitaban los mismos datos de entrada que los códigos originales, normalmente grandes matrices específicas del dominio de la aplicación que dirigían la evolución de los parámetros internos del código
- El usuario final. La información de predicción podía ser examinada desde diferentes ángulos, podía cambiar los parámetros que caracterizaban la simulación para analizar la influencia de cada uno, o simular distintos entornos de ejecución.

1.3. Modelos de programación paralela

En algún momento la computación paralela sustituirá a la computación secuencial. Si hasta ahora no ha sucedido es porque la tecnología de desarrollo de programas paralelos no está suficientemente madura. Hay un consenso en que, para que esto suceda de forma natural, debe aparecer un modelo de computación paralela que sea igual de útil que el modelo von Newmann lo ha sido en secuencial [207].

A este modelo se le pide (1) que introduzca una metodología de desarrollo del código paralelo, que permita inferir una visión detallada de la ejecución del código paralelo; (2) que sea independiente de la arquitectura; (3) que sea fácil de entender por los usuarios; (4) que sea fácil de programar, escondiendo los detalles de ejecución paralela del código en la descomposición del problema en los diferentes niveles de paralelismo, el mapeo a las diferentes unidades de procesamiento y comunicaciones entre ellas; (5) que sea posible inferir el coste de la ejecución de los programas paralelos, donde influye necesariamente la arquitectura del sistema y la red de interconexión: capacidades de los nodos, topología de las comunicaciones y problemas de congestión.

Este modelo de programación paralelo ideal tiene objetivos contradictorios. Pero se han desarrollado distintos modelos de programación con algunas de estas características. Los más importantes son la programación por pase de mensajes, programación lógica y funcional, reducción de grafos, flujo de datos, varios tipos de memoria virtual compartida con coherencia cache, paralelismo de datos, PRAM, BSP, LopG, etc.

1.3.1. Programación por pase de mensajes

En este modelo de programación, una colección de procesos, escritos en un lenguaje secuencial extendido por llamadas a una librería de comunicaciones, envían mensajes para intercambiar datos con otros procesos necesarios para la computación del algoritmo.

Varias librerías implementaron este modelo. Desde los años 90 se estandarizó en el interfaz MPI que está presente en la mayoría de las arquitecturas paralelas distribuidas [97][179]. Soporta comunicaciones síncronas, asíncronas, punto a punto, colectivas, y las últimas versiones pueden incluir información de topologías para optimizar su uso.

1.3.2. Programación sobre memoria compartida

En este modelo, distintas directivas de paralelización de alto nivel se utilizan para indicar operaciones de redistribución de datos y la sincronización entre los procesos. Todos los procesos comparten el mismo espacio de memoria, por lo que desaparece la necesidad de comunicaciones explícitas en el intercambio de datos. Actualmente, OpenMP [61][3] es el estándar de este modelo de programación.

1.3.3. Paralelismo de datos

El modelo de programación por paralelismo de datos se refiere al paralelismo obtenido en los casos en que una misma operación se ejecuta simultáneamente sobre un conjunto de datos. Un programa es una secuencia de estas operaciones [127].

Con este modelo de programación, el programador es responsable de definir la descomposición de las estructuras de datos del algoritmo en el dominio del problema, que son distribuidas por el sistema al inicio de la ejecución, y el compilador es el encargado de particionar las computaciones entre elementos de computación. Cuanto mejor definidas resulten las estructuras de datos, para lo que se utilizan elementos específicos del lenguaje como nuevos tipos de datos, constructores paralelos de tipos conocidos o directivas de distribución de datos, más sencillo resultará al compilador optimizar la ejecución del algoritmo. Éste debe localizar y explotar el posible paralelismo de las operaciones realizadas en las estructuras de datos distribuidas.

Lenguajes como HPF [96], Fortran D [107] o pC++ [43] implementan este tipo de paralelismo. Es un modelo de programación de alto nivel, donde el usuario no necesita especificar el esquema de comunicaciones explícitamente, sino que son generadas por el compilador a partir de la descomposición de las estructuras de datos del problema hecha por el usuario. No todos los algoritmos son susceptibles de ser expresados con este modelo de programación.

1.4. Análisis de rendimiento

El reto principal de la computación paralela es aprovechar el máximo potencial de un sistema con más de un procesador. Con el uso correcto de los recursos se pueden tratar problemas de mayor tamaño. En arquitecturas paralelas esto implica tanto un conocimiento detallado de la arquitectura como el uso de entornos de programación adaptados. Si bien el coste de un código secuencial depende de las capacidades de la máquina y el tamaño del problema, en un código paralelo influye también el número de elementos de procesamiento, sus diferencias en capacidad de computación y el volumen de comunicaciones generado por el algoritmo. Es decir, que un programa paralelo no puede ser evaluado de manera aislada de la arquitectura paralela donde se aplica y el estado en que se encuentra.

Todo el esfuerzo dedicado al desarrollo de la aplicación paralela se ve desperdiciado si el rendimiento del conjunto no cumple con los requisitos de eficiencia esperados para la instancia

del problema. Para reducir estas situaciones, herramientas como el análisis de rendimiento sirven para evaluar y predecir el comportamiento de la aplicación en futuras ejecuciones en el sistema bajo condiciones controladas y planificar la utilización de los recursos hardware, guiando el desarrollo. Pero también identifica carencias y debilidades del sistema, derivadas de la propia arquitectura.

El aspecto del comportamiento de una aplicación en una arquitectura paralela que más interesa es el coste computacional de su ejecución, porque optimizarlo implica directamente una mejora en la productividad. El análisis del rendimiento debe ofrecer información suficiente al usuario para permitir mejorar el comportamiento de la aplicación en el sistema paralelo. Entre las magnitudes que pueden ser usadas para evaluar algún aspecto del rendimiento, la métrica más importante en el análisis es el tiempo de ejecución, seguida del número de FLOPs. Otras alternativas no son tan cómodas de manipular. En los sistemas paralelos también es habitual utilizar métricas como el *speedup* o la eficiencia para caracterizar la escalabilidad del conjunto y cuantificar los beneficios de la paralelización. Estas magnitudes se expresan en función de características relevantes de la arquitectura y el problema, relacionados con la cantidad de paralelismo que podemos explotar en los diferentes niveles disponibles.

Debido a la complejidad inherente de los sistemas paralelos, el análisis de rendimiento es una herramienta fundamental. La ejecución en sistemas paralelos lleva asociada una variedad de ineficiencias que deben ser cuantificadas de manera adecuada. En una ejecución típica, junto con el coste computacional básico, equivalente al de un programa secuencial que resolviera el mismo problema, el programa paralelo sufre pérdidas de rendimiento debido a:

– Comunicaciones entre procesos. Cualquier programa paralelo no trivial requiere que sus nodos se comuniquen datos. El tiempo empleado en transmitir esta información es la principal fuente de ineficiencia paralela.

– Tiempos de espera. Los procesadores paralelos pueden estar esperando por distintos motivos. En muchas aplicaciones es imposible predecir con precisión el tamaño adecuado con el que subdividir el conjunto de tareas que se ejecutan y de manera que se mantenga una carga de trabajo uniforme en todos ellos. Si el problema se descompone de forma ineficiente, se obtiene un mal balanceo de la carga entre los procesos. Lo mismo sucede si los procesadores están demasiado ocupados y no aceptan nuevas asignaciones de tareas, ya preparadas para ser ejecutadas. En otras ocasiones, los procesadores

deben sincronizarse en ciertos puntos del algoritmo para intercambiar resultados parciales y, si no todos están preparados al mismo tiempo, se generan tiempos de espera. Por último, puede que el algoritmo original no sea completamente paralelizable y alguna sección deba ejecutarse en secuencial. Si no se puede solapar con otras computaciones, se convierte en un cuello de botella para el conjunto.

– Computaciones redundantes. Cuando los mejores algoritmos secuenciales para el problema no pueden no ser paralelizados, se utilizan otros algoritmos paralelizables menos eficientes. En estos casos, hay una diferencia en cálculos entre el mejor secuencial y el mejor paralelo. Incluso usando un mismo algoritmo, el paralelo puede necesitar recalcular varias veces información que no esté disponibles en el mismo nodo, pero que el secuencial reutiliza.

No hay una metodología única para realizar un análisis completo de una aplicación paralela en un sistema paralelo, sino que se aplican distintas técnicas que se centran en aspectos concretos del rendimiento.

1.5. Caracterización del rendimiento

Hay tres métodos básicos que se emplean para caracterizar el rendimiento de un sistema computacional cuando ejecuta una aplicación [230], y que se pueden combinar: las mediciones directas, las simulaciones y los modelos analíticos.

1.5.1. Medida directa

La forma más efectiva de conocer el rendimiento de una aplicación particular en un sistema es ejecutar, de hecho, la aplicación en el sistema. Pero los resultados caracterizan exclusivamente esa ejecución particular. La generalización de los datos obtenidos está limitada sólo a las condiciones de ejecución utilizadas y a la configuración concreta del sistema, y no proporciona referencias inmediatas al comportamiento de la aplicación bajo otras condiciones o sistemas. Sin embargo, desde un punto de vista subjetivo, basado en la experiencia y habilidad del usuario, a veces es posible extrapolar estos resultados a las condiciones de ejecución bajo sistemas similares.

Cuando se dispone de acceso al sistema real, este método se utiliza como un mecanismo de detección de posibles problemas de rendimiento durante el desarrollo de las aplicaciones, apoyado y complementado por herramientas interactivas de visualización para descubrir e interpretar relaciones no triviales en los datos de rendimiento [203][112][172].

Un elemento importante para caracterizar la arquitectura en vez de la aplicación de usuario son los *benchmarks*. Son un conjunto de aplicaciones, reales o sintéticas, representativas de una categoría de problemas. El rendimiento de los componentes del *benchmark* proporcionan una caracterización del potencial del sistema y un mecanismo efectivo para la comparación con otros similares. Sin embargo, aunque proporcionan una estimación realista de la eficiencia del sistema, no son referencias válidas para predecir el comportamiento de una aplicación determinada que no esté incluida en el propio conjunto del *benchmark* [78][23][228].

1.5.2. Simulación

Se toman medidas de la aplicación funcionando sobre un software que replica el comportamiento de los distintos elementos que forman el sistema real. Este método proporciona un entorno controlado para tomar datos de rendimiento con mayor control y replicabilidad y sin intrusismo en el sistema real. Es necesario que el sistema sea suficientemente fiel al original para que los resultados sean realistas, pero también suficientemente sencillo para evitar un elevado coste de desarrollo y obtener tiempos de ejecución razonables [180].

Se utiliza para el diseño de nuevas arquitecturas, ya que permite estimar valores de rendimiento de aplicaciones en sistemas que aún no se han construido.

Hay dos tipos de simuladores [136]. Los simuladores por código utilizan el propio código de la aplicación para controlar la simulación, al igual que hace un sistema real. Aunque los resultados son muy precisos, el coste de usarlos es prohibitivo en simulaciones largas y complejas. Los simuladores por traza reutilizan una traza obtenida de la ejecución de la aplicación sobre un sistema real suficientemente similar al simulado [75][232][21][120].

1.5.3. Modelos analíticos

En ellos se caracteriza la aplicación utilizando expresiones analíticas, de manera que el rendimiento pueda evaluarse, cuantitativamente o cualitativamente, de forma independiente al acceso a la arquitectura en que será ejecutado. Se obtiene gran flexibilidad, a costa de tener

que realizar suposiciones y simplificaciones sobre el sistema para poder extraer estos modelos, lo que reduce la precisión del resultado [14][184][44].

Estos modelos son una abstracción que permite describir el conjunto de la aplicación, los sistemas paralelos y las relaciones entre ellos, no de forma exhaustiva, pero sí las características destacadas de los mismos, y dejar que la interacción entre los componentes represente el sistema en su totalidad. Para distintos usos se puede diferenciar entre modelos de arquitectura y de aplicación. Los modelos de arquitectura definen máquinas abstractas con interacciones entre sus componentes representadas mediante parámetros escalares. Los modelos de aplicación describen el comportamiento de aplicaciones al interaccionar con el sistema donde se ejecutan, lo que establece la descripción de la aplicación en función de los parámetros de la arquitectura y permite que el modelo sea aplicable en las arquitecturas que coincidan con la descripción de la máquina abstracta.

Parámetros de entrada

Para el desarrollo de los modelos analíticos se debe disponer de un volumen importante de información del rendimiento real de la aplicación en estudio bajo diferentes configuraciones de entrada.

Los procesadores actuales incluyen contadores hardware que permiten extraer esta información especializada de rendimiento con bajo coste, y que son accesibles a través de librerías externas. Se pueden introducir, en el código de la aplicación, llamadas a estas librerías para acceder, procesar y almacenar estos valores durante la ejecución.

La forma en que estas librerías recogen información de rendimiento puede ser por muestreo. En ese caso, un contador externo a la aplicación, de forma periódica, registra el contexto de la aplicación en ese instante de muestreo. Al terminar, se dispone de un histórico de los diferentes contextos, pero no de información del comportamiento del programa entre cada dos de esos instantes. Este método tiene la ventaja principal de que depende de una temporización externa, por lo que el grado de intrusismo es mínimo. Y la desventaja de que requiere tiempos de ejecución largos para obtener una distribución representativa de la ejecución, o, alternativamente, muestreos más frecuentes.

Otras librerías utilizan un muestreo accionado por evento, que permite registrar la ocurrencia de eventos en colaboración con el usuario, quien especifica aquellos en los que está interesado.

La forma más común de hacerlo es instrumentalizando el código. Es decir, modificando el código de la aplicación para añadir explícitamente, en puntos seleccionados, instrucciones específicas de señalización para la librería de muestreo, y que permiten identificar acciones específicas sobre el código que influyen en métricas y parámetros del rendimiento. La ventaja principal es un mayor control. Pero tiene la desventaja de mayor intrusismo del proceso de medida sobre el comportamiento del sistema [7][47][204].

Esta instrumentalización del código puede realizarse a diferentes niveles. Por ejemplo:

- Desde el código fuente, con un lenguaje de alto nivel, de forma manual o automática. Es la más frecuente, y proporciona una mayor flexibilidad. Pero requiere recompilar el código y tiene implicaciones en el coste de ejecución [156].
- Desde el código fuente con traductores de código, que utilizan directivas del compilador que el preprocesador convierte en instrucciones de código fuente instrumentalizadas. Se obtiene una mayor sencillez en la gestión de los puntos de medida, pero todavía necesita recompilar el código [169][41].
- Una librería accesible en tiempo de ejecución se sustituye por una librería equivalente preinstrumentalizada. Ofrece simplicidad de la instrumentalización, pero requiere un filtrado posterior de datos para seleccionar los relevantes para la aplicación [97].
- En tiempo de ejecución. El entorno de ejecución, una máquina virtual, se encarga automáticamente de hacer la instrumentación de los eventos que aparecen. El intrusismo es mínimo, pero los datos son poco relevantes porque falta información de alto nivel y se vuelve complicado relacionarlos con el comportamiento de la aplicación [51].

Independientemente de la técnica utilizada, se genera un gran volumen de datos de rendimiento, en forma de valores de métricas y parámetros, del que se extraen dos tipos de producto: los perfiles y las trazas de ejecución[1]. El perfil es un resumen estadístico de los valores de rendimiento. En cada instante se registran y se unen a los anteriormente registrados para obtener un valor global instantáneo. Permite obtener una visión general del rendimiento, y reducirlo a unos pocos valores, pero se pierde la dimensión temporal. Las trazas registran la secuencia

[1] En un sistema paralelo, estos valores se calculan para cada elemento de computación.

temporal de valores de rendimiento en cada instante de medida. De esta forma se puede reproducir el comportamiento de estos valores durante la ejecución de la aplicación. El volumen de datos que se maneja con las trazas es elevado, y se han desarrollado herramientas especializadas para filtrar, comprimir, y visualizarlos. La visualización de las trazas permite un análisis visual sencillo de la evolución de la ejecución de la aplicación [79][22][158].

Los perfiles y las trazas se generan habitualmente al terminar el programa, cuando ya no hay más código instrumentalizado en ejecución. Sin embargo, hay herramientas de monitorización en tiempo real que permiten una evaluación instantánea del rendimiento, durante la ejecución, para permitir un ajuste dinámico de los parámetros del sistema y optimizar el comportamiento en la arquitectura. Esto es más habitual en aplicaciones con largos tiempos de ejecución [79][196][173][53].

1.6. Métricas

Al usuario de una aplicación paralela le interesa reducir el tiempo de respuesta o de ejecución del programa. Se asocia de manera natural un mejor rendimiento a las aplicaciones que, haciendo la misma tarea, consuman menos tiempo. Es importante considerar el rendimiento en función del tamaño del problema o del consumo de memoria o de energía. En este apartado se considera sólo la dimensión temporal del rendimiento.

El tiempo de ejecución de una aplicación en un sistema paralelo depende no solo de las particularidades del problema, sino también del número de procesadores disponibles para ejecutar simultáneamente, de las características computacionales de las unidades de procesamiento, de las características de los enlaces de comunicación, de las dependencias entre procesos y de la cantidad de la información intercambiada entre ellos. En principio, para un mismo tamaño del problema, aumentar el número de procesadores asignados podría suponer un aumento del volumen de comunicaciones, y encontrar un balance entre ambos términos es un problema complejo. Por tanto, el tiempo de ejecución paralelo está relacionado principalmente tanto con aspectos del cálculo como de las comunicaciones.

En sistemas paralelos se utilizan las siguientes métricas para evaluar la adaptación del programa a la plataforma.

Granularidad

La granularidad es una medida de la proporción entre tiempo de computación y tiempo de comunicación. Las tareas de granularidad fina tienen un tiempo de ejecución, relativamente, reducido entre comunicaciones, terminan enseguida y devuelven un resultado. En las tareas de mayor granularidad las comunicaciones son poco frecuentes y van a producirse después de largos períodos de cómputo.

Aceleración

La aceleración, o *speedup*, es una medida directa del rendimiento del código, relacionada con el beneficio que supone en el sistema ejecutar la aplicación paralela frente a su versión secuencial. Cuanto mejor paralelizada esté la aplicación, lo que depende del problema, de la implementación y del sistema donde se ejecuta, mayor *speedup* se obtiene. Viene dada por:

$$s = \frac{\text{tiempo ejecución secuencial}}{\text{tiempo ejecución paralelo}} \tag{1.1}$$

Hay diferentes formas de considerar el tiempo de ejecución secuencial y cada una produce una diferente definición. En el *speedup* relativo, el tiempo secuencial es el tiempo en que tarda en ejecutarse el código paralelo en un único procesador. La alternativa es usar el tiempo en que tarda en ejecutarse el mejor programa secuencial posible.

El tiempo de ejecución paralela depende de múltiples parámetros: características del problema, características del sistema y características de la red de comunicaciones. Normalmente se emplea una versión simplificada del *speedup* donde se consideran sólo las relaciones más evidentes, como son el tamaño del problema y el número de procesadores. De esa manera se obtiene un *speedup* parametrizado por estos dos valores: *s(tamaño problema, numero procesadores)*.

Eficiencia

La eficiencia es el porcentaje de *speedup* obtenido en el algoritmo por procesador. Su expresión es la siguiente:

$$E = \frac{s}{p} = \frac{\text{tiempo ejecución secuencial}}{p \cdot \text{tiempo ejecución paralelo}} \tag{1.2}$$

El valor más bajo se obtiene cuando el proceso presenta una aceleración muy baja con muchos procesadores. Observamos que si $s \to 0$, $E \to 0$. El valor más elevado, cuando el proceso es puramente paralelo, $s \to p$ y $E \to 1$.

Redundancia

La redundancia es el inverso del *speedup* relativo. Proporciona una forma de comparar entre el paralelismo software y hardware. Tiene la expresión siguiente:

$$R(p) = \frac{\text{tiempo ejecución paralelo(p procesadores)}}{\text{tiempo ejecución paralelo(1 procesador)}} \tag{1.3}$$

Utilización

La utilización es el porcentaje de recursos que se utilizan durante la ejecución del programa paralelo, y se relaciona con las dos anteriores de la siguiente forma:

$$U = R \cdot E \tag{1.4}$$

Escalabilidad

La escalabilidad es el cambio en el rendimiento cuando se modifican las características de alguno de los parámetros sobre los que se calculaba. Es habitual considerar la escalabilidad de la aplicación al aumentar el número de procesadores sobre los que se ejecuta, ya que es el parámetro más fácil de corregir en un mismo sistema paralelo.

De esta forma, un sistema es escalable si mantiene constante la eficiencia al aumentar el número de procesadores, para tamaños de problema fijos y suficientemente grandes. Es decir, es escalable si cuando aumenta el número de procesadores, disminuye en la misma proporción la fracción de tiempo utilizada para el cálculo paralelo.

La escalabilidad es una magnitud que se puede definir para algoritmos, pero también para arquitecturas, a través de *benchmarks*. Normalmente, todos los sistemas presentan un determinado número de procesadores a partir del cual la eficiencia disminuye de forma notable. Un sistema es más escalable que otro si es mayor el número de procesadores donde comienzan a hacerse evidentes los problemas de eficiencia. La eficiencia de los sistemas disminuye en parte debido a las características de la red de interconexión, ya que, ante la congestión de la red, los tiempos de ejecución aumentan.

1.7. Límites

La adaptación de un código secuencial no trivial a una plataforma paralela tiene su eficiencia limitada por las características de la interacción entre los distintos procesos en que se divide la tarea principal. Expresado en términos de las métricas anteriores, el valor de la eficiencia es (casi siempre) inferior a la unidad.

1.7.1. Ley de Amdahl

La Ley de Amdahl ofrece un cálculo inmediato para el límite teórico de aprovechamiento óptimo del sistema paralelo. Debido a la presencia de distintos procesos colaborando para ofrecer una solución global, en una aplicación paralela, aumentar el número de procesadores que se utilizan no implica un aumento proporcional del rendimiento. La aceleración máxima posible está relacionada con la fracción del código paralela frente a la fracción de código secuencial, y estará limitada en el límite superior por el número de procesadores. Se puede formular del siguiente modo:

$$s = \frac{1}{((1-f)+f/p)} \tag{1.5}$$

donde s es la aceleración conseguida, f es la fracción de la porción paralelizada frente a la secuencial y p es el número de procesadores [19].

En el límite superior $s \to p$, cuando $f \to 1$, es decir, cuando la aplicación es paralelizable completamente, que es equivalente a tener varias instancias de un proceso que no se comunican entre sí. En el límite inferior $s \to 1$, cuando $f \to 0$, es decir, cuando la aplicación es secuencial, y resulta independiente del número de procesadores que le asignemos.

De forma alternativa se puede expresar como:

$$s = \frac{p}{(1+c\cdot(p-1))} \tag{1.6}$$

donde $c = (1-f)$ es la fracción secuencial del código.

1.7.2. Ley de Gustafson

La Ley de Gustafson se aplica a otro tipo de problemas escalables. En ordenadores masivamente paralelos la Ley de Amdahl tiene poca relevancia para casos prácticos. Generalmente, en estos sistemas, se aprovecha el aumento de capacidad de cálculo para aumentar el tamaño

del problema por parte del usuario, controlando la resolución con la que se trabaja, número de etapas del algoritmo, complejidad de las operaciones o cualquier otro parámetro de ajuste que permita obtener una solución en un tiempo razonable. No es tan importante mejorar el tiempo de ejecución como la escala del problema. Por tanto, en estas máquinas es más realista considerar que el tiempo de ejecución es constante y que el tamaño del problema varía.

Las ecuaciones resultantes son similares al caso anterior:

$$f(n) + c(n) = 1 \tag{1.7}$$

donde n es el tamaño del problema, en el parámetro que mejor lo describa, f la fracción paralela del código y c la fracción secuencial.

Para p procesadores, de forma simplificada, se obtiene lo siguiente:

$$s = c(n) + p \cdot f(n) \tag{1.8}$$

Se observa que si la fracción del problema procesada secuencialmente disminuye cuando se aumenta el tamaño del problema, es decir, $c(n) \to 0$ cuando $n \to \infty$, el *speedup* se vuelve proporcional a la fracción paralela, es decir, $s \to p$ cuando $n \to \infty$ y $f(n) \to 1$. Expresado de otra forma, cuando aumenta el problema, la parte secuencial es cada vez más irrelevante, de manera que la parte paralela asume un comportamiento lineal con el número de procesadores, en una aplicación perfectamente paralelizada.

1.7.3. Ley de Sun y Ni

Esta Ley se aplica a problemas escalables limitados por la capacidad de la memoria. Es similar a las leyes de Amdahl y de Gustafson, pero en lugar de mantener fijo el tamaño del problema o el tiempo de ejecución, fija la cantidad máxima de memoria disponible para el problema.

La ley se establece bajo la suposición de que, al aumentar la memoria disponible, por ejemplo aumentando el número de procesadores o la capacidad de cada uno, aumenta también la carga de trabajo asignada a cada proceso. Siendo $G(p)$ el incremento de la carga al aumentar la memoria p veces, se obtiene la siguiente expresión:

$$s = \frac{(c + G(p) \cdot f)}{(c + G(p) \cdot f/p)} \tag{1.9}$$

Se identifican tres casos especiales:

- $G(p) = 1$ corresponde al caso donde el tamaño del problema es fijo, de manera que la ecuación se simplifica a la Ley de Amdahl.
- $G(p) = p$ corresponde al caso donde la carga se incrementa en la misma proporción que la memoria, que corresponde con la Ley de Gustafson con un tiempo de ejecución fijo.
- $G(p) > p$ corresponde al caso donde la carga se incrementa más que la memoria y, en este caso, probablemente se genera un *speedup* superior al caso anterior.

1.7.4. Superlinealidad

La Ley de Amdahl ofrece un límite superior al *speedup* por el número de procesadores:

$$\text{mejor tiempo paralelo} = \text{tiempo ejecución secuencial}/p \tag{1.10}$$

Hay ocasiones donde, por el propio diseño del algoritmo, el algoritmo paralelo en conjunto hace menos trabajo que el algoritmo secuencial y se obtiene *speedups* que resultan superiores a p. Estas situaciones se conocen como casos superlineales. Los procesadores actuales incorporan técnicas para mejorar el rendimiento secuencial: varios niveles de cache, ejecución segmentada en *pipelines*, *hyperthreading*, ejecución fuera de orden, ejecución especulativa, etc. El fenómeno de la superlinealidad aparece cuando el código paralelo puede explotarlas de manera más eficiente, porque, al dividir el problema secuencial, mejora la localidad espacial y temporal del código en cada nodo.

1.8. Modelos analíticos de programas paralelos

Un modelo analítico para un sistema paralelo debe reflejar el coste de ejecución del código paralelo, simplificando y ocultando detalles del sistema real, para permitir comparar la implementación de distintos algoritmos ante el mismo problema y seleccionar el mejor en términos de las métricas mostradas. Es, por tanto, una herramienta orientada al diseño, análisis y evaluación del rendimiento de algoritmos paralelos.

La precisión obtenida en el modelo depende del nivel de abstracción considerado. No solo se necesita que sea suficientemente detallado para reflejar de forma realista aspectos importantes

del sistema, sino también suficientemente abstracto para que sea independiente de la máquina concreta y, además, sencillo de evaluar.

Los modelos analíticos paralelos más utilizados se centran o bien en modelar máquinas paralelas ideales o bien la red de interconexión.

1.8.1. Modelos PRAM

La familia de modelos PRAM hace referencia a una máquina paralela con p procesadores que ejecutan el mismo código de manera síncrona sobre una memoria global de acceso aleatorio compartida. Cada procesador puede acceder a cualquier dirección de memoria en un tiempo unidad. Los diferentes tipos de PRAM se especializan en problemas de contención en los accesos simultáneos para leer y escribir en memoria [95][143].

Estos modelos proporcionan una estimación sencilla del grado de paralelismo que se puede esperar de los códigos, pero ignoran completamente los costes de la red de interconexión.

Distintas extensiones de estos modelos intentan mejorar la aplicabilidad del modelo definiendo reglas para resolver los conflictos de acceso a memoria mediante serialización, como los CRCW PRAM (*Concurrent Read Concurrent Write PRAM*), los EREW PRAM (*Exclusive, Exclusive*) y los CREW PRAM (*Concurrent, Exclusive*). Otras variaciones intentar incorporar características más avanzadas, como latencias, modos asíncronos, fallos de acceso o modelar la congestión.

1.8.2. Modelos basados en la red de interconexión

Estos modelos exponen detalles simplificados de la arquitectura de la red de conexión en la evaluación del sistema [154].

Se ha mostrado en la sección anterior que la comunicación entre procesos limita el rendimiento de las aplicaciones paralelas. Por tanto, el modelo debe poder informar sobre el tiempo empleado en esa tarea. Pero también debe permitir analizar factores de alto nivel, como los problemas de control de flujo y de congestión de red.

Estos modelos permiten enfrentarse a problemas de red. Normalmente el rendimiento de las redes de interconexión en sistemas paralelos con una tasa de tráfico de datos uniforme, sin llegar a la condición de saturación o a problemas de contención, es una función lineal del

volumen de datos que fluyen por la red. En presencia de congestión de la red, los patrones de tráfico no uniformes dan lugar a una degradación del rendimiento. Estos conflictos de la red de interconexión, o bien dependen de algoritmos de encaminamiento deterministas, que responde siempre con degradación ante una misma combinación de mensajes de saturación, y se minimizan utilizando algoritmos adaptativos y caminos redundantes, de ser posible, o bien son resultado de una implementación deficiente de los códigos paralelos que producen fuertes desbalanceos en el patrón de tráfico hacia nodos concretos de la red.

El modelo postal

Este modelo [24] considera un sistema con p procesadores, cada uno de ellos con la posibilidad de usar simultáneamente un puerto de entrada y uno de salida, por lo que pueden enviar y recibir al mismo tiempo un mensaje de tamaño fijo. Un mensaje enviado en un instante t, con una latencia l, que incluye el efecto de empaquetar el mensaje y desplazarlo por la red, será recibido en el destino en el instante $t+l-1$. Cuando $l=1$, el modelo se reduce a la situación de nodos completamente conectados.

El modelo de Hockney

Este modelo asume que el tiempo necesario para enviar un mensaje de tamaño m entre dos nodos se comporta linealmente, y es caracterizado mediante dos parámetros de la red [129]. La expresión es:

$$t = a + b \cdot m \tag{1.11}$$

siendo t el tiempo de comunicación, a la latencia y b el recíproco del ancho de banda.

En este caso los costes de la red de interconexión se centran en un mensaje único e ignoran las interacciones entre múltiples mensajes, como el solapamiento o la congestión de la red.

Los modelos BSP

Esta familia modela una máquina con p procesadores, cada uno con una memoria local, unidos mediante una red de interconexión que permite mensajes de tamaño fijo m. Las computaciones se dividen en pasos, donde en cada uno de ellos el procesador trabaja sobre datos locales y envía o recibe mensajes. Un mensaje enviado en un paso llega en el siguiente y requiere una comunicación global. Cada paso implica una sincronización global de todos los procesadores [206].

Las comunicaciones se modelan con dos parámetros: un parámetro g asociado al ancho de banda y una latencia l asociada al coste mínimo de enviar un paquete por la red, lo que implica un cambio de paso y una sincronización global [225].

El modelo LogP

Este modelo utiliza máquinas básicas conectadas a una red caracterizada por cuatro parámetros: (1) latencia de comunicación, límite superior para enviar un mensaje unidad por la red en ausencia de problemas; (2) *overhead*, tiempo empleado por un procesador para tareas de envío/recepción y que no puede utilizar para cómputo; (3) *gap*, límite inferior para enviar dos mensajes consecutivos por el mismo procesador; y (4) inverso del ancho de banda. La capacidad de cálculo se modela a partir del número de procesadores presentes en el sistema [68].

LogP es un modelo sencillo que sirve como base para otros [142][198]. Los parámetros del modelo son independientes de arquitecturas concretas y permiten el desarrollo de algoritmos portables, obviando detalles específicos de implementación. El principal inconveniente es que está definido para un tamaño de mensaje fijo y no contempla mensajes de longitud variable [20][87][66][82][133][212].

El modelo LogGP

El modelo LogGP extiende al modelo LogP para modelar las comunicaciones de mensajes grandes [16]. Añade como parámetro el coste del envío de una unidad de mensaje, 1 byte, para mensajes grandes. Es una generalización de LogP donde latencia y *overhead* son constantes, pero el *gap* depende linealmente del tamaño del mensaje [29]. El resultado se ajusta con comodidad al comportamiento de las redes de interconexión habituales.

El modelo pLogP

El modelo pLogP es una extensión de LogP para sistemas de área amplia y mensajes de longitud arbitraria, con la finalidad de obtener tiempos estimados de comunicaciones más precisos [144]. Añade varios parámetros adicionales que están en función al tamaño del mensaje: (1) *overhead* del envío; (2) *overhead* de recepción; y (3) el intervalo entre mensajes. En este modelo el tiempo del mensaje en la red está acotado por límites máximos y mínimos de la red física y no puede ser formulado con una expresión analítica.

1.9. Herramientas

Las etapas básicas para el desarrollo de una aplicación en un sistema paralelo se repiten cíclicamente en una serie de pasos, hasta alcanzar el rendimiento esperado. Corresponden a (1) el diseño e implementación de la aplicación, (2) la ejecución con la monitorización necesaria para generar un fichero de traza, y (3) el análisis de las trazas utilizando alguna herramienta que extraiga información estadística.

Muchas veces los análisis se complementan con una representación visual de la traza para que sea el propio desarrollador quien identifique, gracias a sus experiencias previas y conocimiento íntimo del código, el rendimiento global esperado y los puntos de ejecución dónde éste se reduce.

En el contexto de la computación de altas prestaciones, se identifica el uso de las siguientes herramientas: herramientas de instrumentación, de diagnóstico y de predicción.

1.9.1. Herramientas de instrumentación

Son librerías utilizadas para generar eventos en la ejecución del programa que permitan medidas puntuales de las métricas de rendimiento o el acceso a los contadores hardware de los procesadores.

PAPI

Proporciona una interfaz y metodología para acceder a los contadores hardware. Es una estandarización, por parte de fabricantes y vendedores, de un interfaz y un conjunto común de eventos relevantes, a través de una librería independiente de la plataforma [47][7]. Está formada por capas, lo que le permite reimplementar la parte dependiente de la arquitectura mientras mantiene los interfaces de usuario.

PCL

Establece una plataforma común para realizar medidas de rendimiento en las máquinas actuales [4][32]. Está formada por una estructura en dos capas, para tener un interfaz de usuario estable mientras implementa el acceso a los contadores de forma dependiente de la arquitectu-

ra. Soporta llamadas anidadas a las funciones de medición, pudiendo dar una visión jerárquica de las mismas.

1.9.2. Herramientas de diagnóstico

Están orientadas a realizar un diagnóstico del comportamiento de cara a detectar problemas de rendimiento, a partir de los resultados de una ejecución. Dentro de ellas se incluyen las visualizaciones de trazas.

TAU

Se trata de un entorno de análisis de rendimiento de sistemas y aplicaciones paralelas [203]. Está formado por un conjunto integrado de herramientas de instrumentación, medida y análisis, que soporta varios lenguajes de programación. Incluye distintos mecanismos de instrumentación, a nivel de código fuente o código compilado, pero también de las comunicaciones mediante pase de mensajes o paralelismo de datos. Permite seleccionar entre distintas métricas temporales y el acceso directo a contadores hardware de los procesadores. Ofrece una herramienta integrada de visualización y permite exportar las trazas para utilizar visualizadores independientes. Por último, integra también una herramienta de minería de datos que permite realizar un análisis transversal sobre múltiples instancias de experimentos, utilizando técnicas de clustering y análisis de correlación, entre otros.

SCALASCA

SCALASCA es un entorno de código abierto especialmente diseñado para el análisis de rendimiento de sistemas paralelos masivos, aunque también es aplicable a sistemas de menor escala [9][112]. Este entorno se centra principalmente en aplicaciones paralelas de carácter científico o de ingeniería. Utiliza instrumentación para obtener perfiles o trazas. Dispone de un mecanismo de análisis post mortem de las trazas obtenidas, que permite la identificación de potenciales cuellos de botella del rendimiento, en particular, de aquellos relacionados con las comunicaciones y las sincronizaciones. La información de perfiles y el resultado de análisis de las trazas puede visualizarse en una herramienta interactiva integrada en el entorno. Además, incluye conversores para exportar las trazas a otros formatos compatibles.

HPCTOOLKIT

El análisis que realiza HPCTOOLKIT se basa en la correlación de las métricas obtenidas dinámicamente durante la ejecución del programa y la estructura del código fuente [6][13]. Para obtener un análisis independiente del lenguaje, la estructura lógica del programa se obtiene directamente a partir del código binario correlacionando las métricas de rendimiento y la estructura lógica de la aplicación. Esto permite la creación de una descripción independiente del hardware para la predicción del comportamiento de la aplicación en otras arquitecturas.

VAMPIR - ITAC

Vampir es una herramienta comercial ampliamente utilizada en HPC. Durante la ejecución de un programa, una herramienta genera una traza que se visualiza interactivamente con Vampir. Instrumenta automáticamente las funciones MPI mediante la técnica de interposición de librería, pero también permite añadir eventos definidos por el usuario utilizando una API para instrumentación de código fuente [172].

ITAC (Intel Trace Analyzer and Collector) es un entorno para la instrumentación y el análisis de códigos paralelos MPI. Este entorno está compuesto por una herramienta de instrumentación (ITC) y una herramienta de visualización interactiva (ITA).

Paraver

Paraver utiliza un formato de traza flexible, que permite realizar diferentes tipos de análisis. Estas trazas pueden obtenerse a partir del código fuente de aplicaciones paralelas escritas en OpenMP o MPI, pero también pueden obtenerse a partir del simulador Dimemas. Ofrece representación visual personalizable de los eventos registrados, y puede realizar un análisis cuantitativo de las diferentes métricas consideradas en la traza, la generación de nuevas métricas derivadas o el análisis concurrente de varias trazas [71][151].

Pablo, svPablo

Está orientado al análisis y visualización de rendimiento, Dispone de una librería de instrumentación para códigos MPI, y un módulo para extraer información de traza del código emitido por compiladores de HPF. Este módulo se ha utilizado para extraer la información de rendimiento de los núcleos de la librería PARAISO [188][195][39].

1.9.3. Herramientas de predicción

Agrupa software que predice el comportamiento de una aplicación al ejecutarla en un sistema determinado, a partir de la caracterización del comportamiento obtenido por ejecuciones reales, inspección del código, simulación o modelos analíticos.

Paradyn

Evalúa el rendimiento de sistemas paralelos y distribuidos. Es fácilmente extensible, y está orientado a problemas temporalmente costosos y sistemas con muchos nodos. Se centra en la búsqueda de problemas de rendimiento mediante un módulo que minimiza la instrumentación sobre el sistema medido, realizándola de forma dinámica al tiempo que se ejecuta la aplicación. Automatiza una gran parte del proceso de búsqueda de problemas de rendimiento, proporcionando, además, herramientas para una visualización interactiva [167].

Dimemas

Es un simulador dirigido por eventos para predecir el comportamiento de programas paralelos que utilicen el paradigma de paso de mensajes [71][151]. Es capaz de reconstruir el comportamiento temporal de la aplicación paralela a partir de un fichero de traza que contiene datos de rendimiento de una ejecución real.

Los eventos registrados en la traza son reproducidos en una máquina virtual, descrita por un conjunto de parámetros arquitecturales y de rendimiento. Las diferentes configuraciones de parámetros permiten la simulación de diferentes tipo de arquitecturas. En particular, la arquitectura de Dimemas se corresponde con una red de multiprocesadores SMP. La red de interconexión está formada por varios buses, que determinan el número máximo de mensajes simultáneos, y varios enlaces con cada nodo SMP. El coste de la comunicación, tanto de mensajes punto a punto como de comunicaciones colectivas, se calcula mediante modelos lineales, aunque se contemplan efectos no lineales como conflictos de red. El resultado de la simulación es una nueva traza que describe el comportamiento de la aplicación sobre la plataforma simulada.

FASE

Es un entorno de simulación que se basa en la separación del problema en dos dominios diferentes: simulación y aplicación [120]. En el dominio de la simulación, el sistema virtual se construye mediante la conexión de modelos abstractos de los componentes individuales. En el dominio de la aplicación, la información proporcionada por perfiles y trazas se utiliza conjuntamente para obtener una caracterización precisa de la aplicación.

WARPP

Está diseñado para analizar el rendimiento de aplicaciones MPI en sistemas MPP mediante la simulación de eventos discretos [123][122]. Este entorno ha sido desarrollado a partir del entorno PACE [175]. Implementa un proceso automático de caracterización de las aplicaciones que genera una descripción de su comportamiento en función de los eventos interpretables por el simulador. Por otro lado, el entorno también implementa un proceso automático que se basa en el análisis de la ejecución de benchmarks paralelos para obtener una caracterización del coste de cada evento en el sistema paralelo sobre el que se realizará la simulación. El simulador de eventos reproduce el flujo de eventos de la aplicación, evaluando el coste de la sucesión de eventos en la arquitectura considerada.

Prophesy

Es un entorno de análisis que permite obtener automáticamente un modelo analítico de la ejecución de aplicaciones a partir de la información de rendimiento obtenida de ejecuciones reales [8][216]. Contiene tres módulos: instrumentación, bases de datos y análisis. El primero permite la instrumentación automáticamente a nivel de código fuente, pero también permite insertar directivas de forma manual. Además de los datos de rendimiento el último módulo construye el modelo utilizando una base de datos de plantillas de modelos y una base de datos de sistemas. Proporciona diferentes métodos de modelado, incluyendo el acoplamiento de kernels independientes.

USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA

CAPÍTULO 2

COMUNICACIONES EN EL GRID

Este capítulo examina la implementación de una librería de pase de mensajes en el Grid. Se inicia con una descripción breve de los principios básicos de este paradigma de programación paralela, en un sistema basado en la plataforma Globus, para, a continuación, pasar a construir un modelo que caracteriza el tiempo de ejecución de las operaciones de comunicación punto a punto y colectivas. El análisis y modelado posterior están diseñados para el software utilizado en el proyecto CrossGrid, en desarrollo entre 2002 y 2005. En la actualidad, existen nuevas versiones del software Grid para las que es posible que el análisis llevado a cabo no sea totalmente aplicable a los sistemas actuales, pero, en cualquier caso, los fundamentos de la metodología desarrollada son válidos.

2.1. Paralelismo por pase de mensajes

De las múltiples tecnologías desarrolladas para aprovechar eficientemente las capacidades de cálculo de arquitecturas paralelas, el paradigma de pase de mensajes es uno de los más antiguos y, a la vez, de los más usados en la actualidad. Sus características principales son:

- Un modelo de memoria distribuida con un espacio de direcciones privado para cada proceso
- Paralelismo explícito. Los procesos colaboran para realizar una tarea computando de forma independiente con sus datos locales e intercambiando mensajes para comunicar

estos resultados parciales. Este intercambio de información se debe incluir explícitamente en el código

La ventaja principal de esta forma de paralelismo es el no tener excesivos requisitos del hardware base, pero, a cambio, exige un mayor esfuerzo de planificación por parte del desarrollador. Éste debe analizar los algoritmos secuenciales originales, identificando cómo descomponerlos en secciones concurrentes de manera eficiente, y programar explícitamente los puntos de intercambio de información necesarios para mantener sincronizado el cálculo entre procesadores al ejecutarlo en un sistema de múltiples nodos.

La gran alternativa a este paradigma es el paralelismo de memoria compartida. En éste, cada procesador tiene la capacidad de acceder directamente a la memoria del resto, y esta característica se utiliza para compartir los resultados parciales de los nodos. Su principal problema es su limitación a arquitecturas paralelas que permitan construir un espacio de direccionamiento global de la memoria. Otras posibilidades para la programación paralela son las extensiones de paralelismo de datos a lenguajes secuenciales clásicos, siendo HPF (*High Performance Fortran*) [96] una de las soluciones que alcanzó mayor popularidad, así como lenguajes basados en paralelismo funcional, como Sisal [92].

Históricamente se han utilizado dos aproximaciones a la implementación de paralelismo por pase de mensajes en una plataforma [227]. La primera es a través de un lenguaje que incluya las directivas básicas para especificar concurrencia en el código, comunicación y sincronización. La segunda implica el uso de librerías de pase de mensajes explícitamente desde lenguajes secuenciales clásicos.

Uno de los lenguajes que siguen la primera aproximación es Occam [132], que incorpora primitivas para especificar la ejecución paralela o secuencial de código, con soporte para aritmética, bloques condicionales y repeticiones mediante bucles, así como el mapeado de procesos a procesadores. Las comunicaciones se realizan de manera síncrona mediante canales unidireccionales con tipo y están soportados los patrones de comunicación básicos punto a punto y uno a muchos condicional. Su sencillez permite probar matemáticamente la corrección del código. Otro ejemplo similar es el lenguaje Fortran M [106], que utiliza canales dinámicos, y está construido a partir el lenguaje estándar Fortran 77 con directivas paralelas para convertirlo en una plataforma de desarrollo más accesible.

Por otra parte, las soluciones basadas en pase de mensajes definen librerías que incorporan llamadas para realizar algún tipo de comunicación punto a punto entre procesos, operaciones colectivas en grupos de procesos y gestión de procesos. Con estas nuevas capacidades, los códigos resultantes tienen disponible un modelo funcional realista de la arquitectura paralela, proporcionando oportunidades para un mayor control sobre la localidad espacial de los datos y de la jerarquía de comunicaciones, lo que permite implementaciones con diferentes niveles de eficiencia.

Soluciones de este tipo han sido desarrolladas por empresas de hardware paralelo específicamente para sus arquitecturas, pero también se han creado implementaciones portables con la colaboración entre industria, organizaciones y centros de investigación. Entre éstas se cuentan: PVM, MPI, Express, PICL, PARMACS, p4 o Zipcode.

Como ejemplo de estas librerías, PVM (*Parallel Virtual Machine*) [213] tuvo gran popularidad en la última década del siglo XX. Permite que una colección de elementos de computación pueda ser accedido como una única máquina paralela para colaborar en la resolución de un problema. Los nodos no tienen que ser homogéneos y gestiona de manera transparente las conversiones de datos. Está pensado como un sistema completo para el desarrollo de computación paralela que incluye (1) la librería de pase de mensajes para comunicación, sincronización y gestión de procesos o tareas, (2) un demonio que se ejecuta en cada nodo para identificar y gestionar los recursos locales, (3) un servicio para gestionar grupos de procesos, y (4) un interfaz de usuario con el que se configura la máquina virtual. En este sistema, un nodo privilegiado es el responsable de la interacción con el usuario y operaciones de entrada/salida de datos. Para resolver un cálculo, éste crea e inicializa en el resto de nodos el conjunto de procesos necesario. En conjunto, PVM se enfrenta a los mismos problemas del Grid, pero a menor escala.

En la actualidad, MPI (*Message Passing Interface*) [97] es la librería de pase de mensajes más utilizada. MPI hace referencia a un estándar propuesto para librerías de pase de mensajes interoperables con las que desarrollar programas paralelos portables en lenguajes secuenciales [121]. El estándar define tanto la sintaxis como la semántica de un conjunto básico de funciones de la librería orientadas a comunicaciones punto a punto, comunicaciones colectivas, gestión de grupos de procesos y soporte para varias topologías de mapeo proceso a procesador. Por el contrario, MPI no define un mecanismo para crear procesos y es responsabilidad del sistema ofrecer el mecanismo para crearlos y ubicarlos (en modo SIMD o MIMD).

Características propias de MPI son el uso de comunicadores para aislar comunicaciones y el uso de tipos de datos abstractos para facilitar la portabilidad; también la posibilidad de definir topologías virtuales con los procesadores. Este estándar está implementado por librerías como MPICH, MPICH-G2, Open MPI, PACX-MPI, LAM, CHIMP-MPI, STAMPI, MagPIe o IMPI, además de desarrollados específicos para hacer compatibles arquitecturas concretas con el estándar (y que conocemos como implementación MPI del fabricante).

2.2. MPI en el Grid

MPICH [121] es una de las implementaciones más populares del estándar MPI. Su objetivo principal es reducir las barreras de entrada para el uso de computación paralela convirtiendo la programación por pase de mensajes en un modelo de programación portable y de alto rendimiento para clusters heterogéneos y sistemas distribuidos [141][102][101].

Fue iniciada como una colaboración entre el Argonne National Laboratoy y la Mississipi State University en el año 94, y ha evolucionado con el estándar. La disponibilidad y eficiencia de MPICH sobre múltiples arquitecturas ha contribuido al éxito de MPI en la comunidad científica.

Tiene soporte para las arquitecturas más frecuentes, como los supercomputadores paralelos de memoria distribuida, donde las comunicaciones MPI reutilizan el hardware propietario de alto rendimiento presente; los supercomputadores paralelos de memoria compartida, donde la librería utiliza características hardware de estas arquitecturas para simular operaciones de pase de mensajes directamente sobre la memoria de los nodos; los clusters de computadores débilmente acoplados, donde la interoperabilidad se apoya en comunicaciones TCP/IP entre nodos de computación; y los sistemas Grid, un caso particular del anterior, donde la librería utiliza los servicios Grid para soportar las particularidades del sistema.

Esta portabilidad deriva de un diseño en capas, que permite a los fabricantes aprovechar de forma inmediata mecanismos específicos de la arquitectura para transferir los datos mientras respeta el interfaz MPI. Con objeto de maximizar esta propiedad, la mayor parte del código de MPICH es independiente de la plataforma. La implementación de las estructuras, objetos y operaciones internas de la librería se realiza en función de elementos portables más simples, abstracciones de datos y macros que forman una especificación denominada ADI (*Abstract Device Interface*) y que ofrece un modelo de comunicaciones mucho más simple que MPI en-

tre la aplicación de usuario y los procesos paralelos. MPICH implementa las funcionalidades del ADI mediante un diseño en múltiples capas con interfaces estándar, que combinan abstracciones de más bajo nivel con funciones dependientes de la arquitectura. Cada adaptación de la librería a una determinada arquitectura reemplaza algunas de las rutinas de la implementación básica dejando intacta la sintaxis de la librería.

Una adaptación de la librería MPICH a sistemas Grid basados en la plataforma Globus Toolkit 2 es MPICH-G2. Se trata de una colección de componentes software diseñados para dar soporte al desarrollo de aplicaciones en entornos de computación de alto rendimiento heterogéneos y altamente distribuidos, de manera que en el conjunto puedan participar simultáneamente varias organizaciones. Permiten ejecutar tanto aplicaciones secuenciales como aplicaciones paralelas MPI [140][102].

La librería oculta la heterogeneidad del sistema utilizando los servicios definidos por esta colección de software para tareas de autenticación, autorización, creación de procesos, migración de ejecutables, planificación, gestión de procesos, ejecución, comunicación, redirección de entradas y salidas, y gestión de recursos para el acceso remoto a ficheros de datos. Como resultado, el usuario puede utilizar nodos en máquinas distribuidas entre diferentes organizaciones utilizando los mismos comandos que utiliza en un computador local:

- Los servicios de información de The Globus Toolkit se utilizan para determinar cómo obtener acceso a los ordenadores seleccionados. Los servicios de seguridad se utilizan para gestionar la autenticación y autorización en cada localización
- Los servicios de ejecución se utilizan para mover el código a los distintos computadores
- Los servicios de gestión de recursos se utilizan para iniciar los procesos en cada computador a través de los planificadores locales
- Los servicios de comunicación se utilizan para gestionar los diferentes métodos de comunicación que aparezcan en el conjunto del sistema, tales como protocolos específicos del fabricante, memoria compartida, implementaciones MPI o comunicaciones TCP
- Los servicios de acceso a ficheros se utilizan para redirigir la entrada y salida estándar a la terminal de usuario independientemente de su localización
- Los servicios de gestión de procesos permiten al usuario monitorizar el progreso de la aplicación y terminarla si fuera necesario

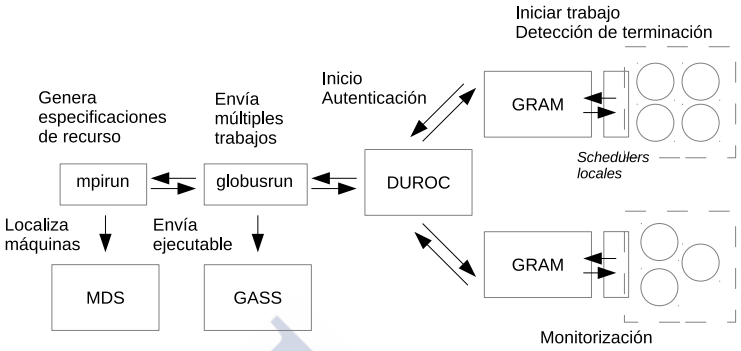

Figura 2.1: Esquema de inicio de una tarea MPICH-G2, mostrando los servicios del Globus Toolkit que ocultan los detalles de implementación de los clusters.

Al mismo tiempo MPICH-G2 también permite explotar la gestión de la heterogeneidad a nivel de aplicación de usuario. Los comunicadores de MPI pueden se utilizados para representar la estructura jerárquica del sistema completo y así adaptar el esquema de las comunicaciones para una mayor eficiencia. Esto permite implementar estrategias de QoS, pero también las operaciones colectivas multinivel conscientes de la topología que veremos más adelante.

El procedimiento básico para la preparación y ejecución de una aplicación con MPICH-G2 sigue los siguientes pasos (figura 2.1):

Antes de iniciar una aplicación MPICH-G2, el usuario debe obtener las credenciales que lo autenticarán en cada nodo del Grid a través de servicios del GSI (*Grid Security Infrastructure*). Una vez autenticado puede seleccionar las máquinas donde ejecutar el código según la disponibilidad de recursos, uso o capacidades de red, interrogando al sistema MDS (*Monitoring and Discovery Service*).

El usuario utiliza el comando estándar `mpirun` para iniciar la aplicación que, en esta versión, requiere ficheros en formato RSL (*Resource Specification Language*) [69][70] para identificar las necesidades del programa y establecer restricciones y parámetros. Un ejemplo de este tipo de fichero se muestra en la figura 2.2. La implementación MPICH-G2 llama al servicio DUROC (*Dynamically-Updated Request Online Coallocator*) para trasladar la aplicación a las máquinas especificadas.

DUROC se apoya en el servicio GRAM (*Grid Resource Allocation and Management*) para iniciar y supervisar el conjunto de procesos paralelos, uno por máquina. Por cada proceso se genera una petición a un servidor GRAM remoto, quien autentica al usuario, realiza la autorización local e interactúa con el planificador local para iniciar la ejecución. DUROC también mantiene los diferentes procesos MPI vinculados a una única tarea.

MPICH-G2 implementa una barrera implícita en la función de inicialización MPI que espera a que todos los procesos sean cargados sobre las máquinas antes de empezar la ejecución. Una vez iniciada la aplicación, elije automáticamente el protocolo de comunicaciones más eficiente entre cada par de procesos. De esta forma, puede utilizar protocolos específicos del fabricante para comunicaciones internas, memoria compartida, o versiones MPI locales; o bien delegar en el dispositivo globus2 para comunicaciones TCP/IP. De ser necesario, también se encarga de hacer conversiones de datos entre diferentes arquitecturas.

Para transferir los ficheros entre la máquina local y los nodos remotos se utilizan los servicios de GASS (*Global Access to Secondary Storage*), ofreciendo una API y comandos de usuario para llevar a cabo la trasferencia remota. En Globus es el usuario el responsable de la transferencia de los ficheros de entrada a las máquinas donde se va a ejecutar la aplicación y la transferencia de los ficheros de salida a la máquina local al finalizar la ejecución. Esta tarea puede automatizarse dentro del fichero RSL.

2.3. Comunicaciones punto a punto

En el modelo de paralelismo por pase de mensajes un programa paralelo se trata como una colección de programas secuenciales que se ejecutan en los nodos del sistema paralelo y se intercambian información con primitivas de comunicación punto a punto.

Para realizar una caracterización del comportamiento del programa hay que añadir, al coste del programa secuencial original y los costes derivados de las comunicaciones. Esto no es inmediato, ya que la evolución de las comunicaciones depende de múltiples características sobre las que no siempre se tiene control, como la semántica del problema, los protocolos software, las características hardware, la topología de la red, el tipo de enrutado y el modelo de gestión del tráfico [153][73]. Estas dependencias se pueden reducir utilizando el modelo de comunicaciones adecuado.

```
+
( &(resourceManagerContact="six03.inv.lugo.usc.es")
   (count=1)
   (label="subjob 0")
   (environment=(GLOBUS_DUROC_SUBJOB_INDEX 0)
                (GLOBUS_LAN_ID 0)
                (LD_LIBRARY_PATH /usr/local/globus/lib/))
   (directory="/home/globus")
   (executable="/home/globus/bcast2")
   (stdout="/home/globus/bcast1c.stdout")
   (stderr="/home/globus/bcast1c.stderr")
)
( &(resourceManagerContact="six05.inv.lugo.usc.es")
   (count=1)
   (label="subjob 1")
   (environment=(GLOBUS_DUROC_SUBJOB_INDEX 1)
                (GLOBUS_LAN_ID 1)
                (LD_LIBRARY_PATH /usr/local/globus/lib/))
   (directory="/home/globus")
   (executable="/home/globus/bcast2")
   (stdout="/home/globus/bcast1c.stdout")
   (stderr="/home/globus/bcast1c.stderr")
)
```

Figura 2.2: El *script* RSL proporciona la sintaxis básica para componer descripciones de recursos complejas, permitiendo que los distintos componentes de gestión introduzcan pares de valores (atributo,valor) en la estructura común.

2.3.1. El modelo LogP de comunicaciones

El modelo LogP es un modelo de la red de comunicaciones para sistemas multiprocesador de memoria distribuida, donde cada nodo se considera un computador completo con su procesador, cache, y memoria, estando los nodos conectados entre sí por una red de comunicaciones, comunicándose a través de mensajes punto a punto. El modelo tiene la gran ventaja de evitar una especificación detallada de la red de interconexión, sustituyendo su influencia por la definición de parámetros de coste asociados al procesamiento de mensajes [68].

Este modelo se define a partir de tres reglas básicas. Primera, las comunicaciones entre procesadores suponen un gran costo con respecto a los accesos a la memoria local. Segunda, las redes de interconexión limitan el ancho de banda por procesador comparado con el ancho de banda de acceso a la memoria local. Y tercera, existe un coste computacional para los proce-

sadores al final de una comunicación entre dos de ellos, que es independiente de la latencia de transmisión entre ellos.

Con estas propiedades, el rendimiento de una comunicación viene caracterizado por los siguientes parámetros (figura 2.3):

- l, límite superior en la latencia o retraso en el que incurre un nodo cuando envía un mensaje a su nodo destino
- o, sobrecoste, definido como el tiempo que un procesador está ocupado en la transmisión o recepción de un mensaje; durante este periodo de tiempo, el procesador no puede realizar otro tipo de operaciones. El modelo diferencia entre os, sobrecoste de envío (el tiempo que tarda un procesador en enviar un mensaje) y or, sobrecoste de recepción, (el tiempo que le lleva a un procesador recibir un mensaje), pero asume que son similares y de coste o
- g, *gap*, definido como el intervalo de tiempo mínimo entre dos transmisiones o recepciones consecutivas de mensajes en un procesador; el inverso de g es al ancho de banda de comunicaciones disponible por procesador
- p, número de nodos en el sistema distribuido; las características del nodo no se especifican en el modelo

l, o y g se expresan en unidades de tiempo y no todos ellos son de igual importancia. El modelo asume que todos los mensajes son del mismo tamaño, pequeño, pero, tal como se ha comentado en la sección 1.8.2, existen versiones modificadas de este modelo para soportar mensajes de gran tamaño que se desplazan en una red de comunicaciones no congestionada. Se pueden tratar problemas de congestión en la red, pero implica introducir información acerca de la topología de red con lo que se pierde la generalidad de este modelo.

2.3.2. El modelo simplificado

El anterior es un modelo de bajo nivel. También tiene demasiados parámetros dependientes de la arquitectura y de las capacidades de la red de conexión que no son monitorizados dentro del proyecto CrossGrid, para el que se ha desarrollado el trabajo presentado en esta tesis. Para evaluar el rendimiento dentro de las limitaciones impuestas por los datos disponibles se debe

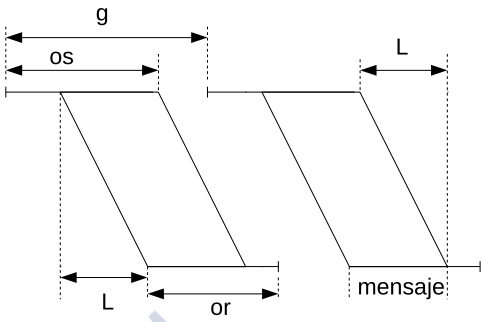

Figura 2.3: Parámetros del modelo LogP: latencia (L), gap (g), sobrecoste de envío (os) y de recepción (or).

trabajar con un modelo más sencillo. En concreto, no interesa tanto representar el solape de mensajes en el tiempo como representar la secuencia lógica de comunicaciones.

Para abordar este problema, al igual que antes, se definen unas reglas básicas que debe seguir el modelo. Primera, que las comunicaciones suceden en pasos discretos. Es decir, un nodo no puede enviar y recibir mensajes al mismo tiempo, un nodo no puede realizar dos comunicaciones simultáneas y no puede recibir simultáneamente dos comunicaciones. Segunda, no hay solape entre envíos, de manera que un nodo que está realizando una comunicación no puede comenzar una nueva hasta terminar con la que está en curso. Tercera, cuando se termina de enviar o recibir un mensaje se puede comenzar inmediatamente con el siguiente, lo que implica que escondemos las particularidades de comunicación de los nodos dentro del propio tiempo de envío del mensaje. Y cuarta, que el coste de las comunicaciones no introduce carga computacional en los nodos.

Con estas características y asumiendo que el tiempo para comunicar dos nodos en el sistema paralelo es la combinación del tiempo para preparar el mensaje para su transmisión y el tiempo que tarda en atravesar la red hasta su destino, hemos desarrollado un modelo sencillo cuyos parámetros son los siguientes:

– *ts*, tiempo de inicialización; es el tiempo de preparación del mensaje en el nodo inicial, incluyendo la construcción de las cabeceras, la información de corrección de errores, la información de enrutado y el tiempo de activar el interfaz entre el nodo y el sistema de enrutado. Este tiempo se contabiliza una vez para cada mensaje

– th, tiempo de salto; es el tiempo desde que el mensaje abandona un nodo hasta que su cabecera llega al siguiente nodo con el que tiene conexión directa; está relacionado con la latencia de los sistemas de enrutado.

– tw, tiempo de transferencia; es el tiempo de transferencia que le lleva al mensaje atravesar un enlace de red y es inversamente proporcional al ancho de banda de los enlaces.

En este modelo un mensaje de tamaño m entre dos nodos separados entre sí por l conexiones emplea en su ejecución un tiempo t, que cumple:

$$t = ts + (th + tw \cdot m) \cdot l \tag{2.1}$$

Sin embargo, un sistema descrito por esta ecuación hace un uso ineficiente de los enlaces entre nodos. Cada mensaje se mueve en pasos discretos entre enlaces, de manera que no puede transmitirse nada del mensaje al siguiente enlace hasta que finalice la recepción del mensaje completo en el actual.

Cuando el mensaje es dividido en paquetes más pequeños y se envía de manera segmentada, se puede demostrar [118] que la ecuación anterior se convierte en la siguiente, donde tw' combina el ancho de banda original con la nueva carga en la red debida al aumento del número de paquetes:

$$t = ts + th \cdot l + tw' \cdot m \tag{2.2}$$

Según esta ecuación, para obtener comunicaciones eficientes en un sistema paralelo las alternativas pasan por minimizar el número de mensajes cortos sustituyéndolos por uno largo (se reduce el número de veces que se contabiliza ts), minimizar el volumen de comunicaciones (se reduce m), minimizar el número de enlaces necesarios para comunicar nodos (se reduce l) o mejorar las características tecnológicas del sistema (se reducen tw' y th).

Este modelo aún se puede simplificar. Para las comunicaciones con la librería MPICH-G2, ts es un parámetro que no puede ser ignorado, ya que agrupa el coste de autorización en los mensajes de usuario, relacionado con la política de QoS y de gestión de recursos en cada organización a lo largo del Grid. Por el contrario, th está dominado por tw para mensajes largos o por ts para mensajes cortos, a la vez que el número de enlaces entre nodos de computación

es muy reducido. Sin excesiva pérdida de precisión, podemos ignorar este parámetro para obtener como ecuación básica:

$$t = ts + tw' \cdot m \tag{2.3}$$

Éste es un modelo válido para un sistema completamente conectado, que admite comunicación directa entre cada pareja de nodos, donde todos los enlaces presentan las mismas características, y con una red de comunicaciones no congestionada. En redes congestionadas, el modelo sigue la misma ecuación sustituyendo el ancho de banda por un ancho de banda efectivo que tenga en cuenta la pérdida de rendimiento debida a la congestión.

Para facilitar el análisis de las comunicaciones, estos parámetros se pueden resumir en una matriz de costes que represente el tiempo de comunicación entre nodos para un tamaño de mensaje fijado, donde las comunicaciones entre cada pareja de nodos pueden tener características particulares, de la siguiente manera:

$$t_{ij}(m) = ts_{ij} + th_{ij} + tw'_{ij} \cdot m \tag{2.4}$$

o, ignorando las latencias:

$$t_{ij}(m) = ts_{ij} + tw'_{ij} \cdot m \tag{2.5}$$

El resultado es un modelo compacto que facilita el análisis de comunicaciones complejas. Los resultados obtenidos pueden ser más adelante interpretados dentro del modelo LogP, si fuera necesario.

Este análisis asume que los parámetros de la red se mantienen estables en el tiempo, al menos durante una cantidad de tiempo superior al tiempo de cada operación de comunicación. En el mundo real aparecen efectos inesperados no despreciables que convierten estos cálculos en aproximaciones teóricas. Por ejemplo, en la referencia [45], la evolución de un algoritmo simple de comunicación en el Grid se desvía de la esperada debido a cargas aleatorias en los nodos por la propia infraestructura de monitorización.

Dentro del proyecto CrossGrid, utilizando los servicios de monitorización centrales, el usuario obtenía información actualizada del estado de la red en términos de ancho de banda y latencia para cada pareja de nodos con la que definir la matriz de costes del modelo simplificado.

2.3.3. `MPI_Send` y `MPI_Recv`

El intercambio básico de información entre nodos en el estándar MPI se realiza mediante comunicaciones punto a punto, donde las operaciones elementales son `send` y `recv`. Los prototipos de estas funciones son:

$$\begin{array}{c} send(*buffer,\ numero\ de\ elementos,\ procesador\ destino) \\ recv(*buffer,\ numero\ de\ elementos,\ procesador\ origen) \end{array} \tag{2.6}$$

donde el buffer en *send* apunta a las posiciones de memoria que almacenen los datos que van a ser enviados, el buffer en *recv* apunta a las posiciones de memoria que van a recibir los datos, *número de elementos* hace referencia a la longitud del mensaje y *procesador destino* y *origen* son identificadores de los nodos que participan en el intercambio. El procesador identificado como origen va a ejecutar la operación `send`, y el procesador destino la operación `recv`.

El tratamiento de esta interacción básica depende de la implementación concreta de la librería MPICH. Muchas plataformas que implementan pase de mensajes disponen de hardware adicional en las interfaces de red y acceso directo a memoria para liberar a la CPU de las tareas de gestión de los buffers. En ese caso, los comandos `send` y `recv` pueden iniciar la transferencia de datos y retomar la ejecución, solapando de manera efectiva computación y comunicación.

En el estándar MPI 1.1 las operaciones básicas `send` y `recv` están disponibles como las funciones `MPI_Send` y `MPI_Recv`. Además, el estándar ofrece varios modos más de comunicación punto a punto que permiten controlar con mayor precisión las relaciones e interacción entre el proceso emisor y el proceso receptor. Cada uno de ellos se implementa con una nueva función `send`. Son los siguientes:

– Modo estándar, función `MPI_Send`. En este modo, es la librería MPI y no el usuario quien decide si los datos se copian a un buffer local antes de ser enviados, liberando al procesador origen de esperar a que el destino comience con la operación de recepción, o si las condiciones actuales de los recursos del sistema recomiendan enviar los datos y dejar al origen esperando a que el destino realice la comunicación. Este modo existe para favorecer la portabilidad del estándar, por lo que se recomienda no depender de buffers del sistema, pero aprovecharlos cuando están disponibles para mejorar la eficiencia.

- *Buffered*, función `MPI_BSend`. El emisor copia los datos a un buffer local y termina, independientemente del estado del destino
- Síncrono, función `MPI_SSend`. El emisor puede comenzar la transmisión independientemente del destino, pero no termina hasta asegurarse de que éste inicia la recepción
- *Ready*, función `MPI_RSend`. El emisor no puede comenzar la transmisión hasta que el destino esté disponible

`MPI_Send` es una función bloqueante, lo que significa que no termina hasta que el proceso es libre de acceder y reutilizar los buffers de datos, ya sea porque han sido copiados al buffer del proceso receptor o a un buffer del sistema local. Lo mismo sucede con `MPI_BSend`, `MPI_SSend` y `MPI_RSend`. `MPI_Recv` es también bloqueante. Cuando termina, el buffer contiene los datos de la comunicación definitivos.

Todas estas funciones tienen también una versión no bloqueante: `MPI_ISend`, `MPI_IBSend`, `MPI_ISSend`, `MPI_IRSend` y `MPI_IRecv`. En ellas la función termina antes de que la operación se complete y antes de que el usuario tenga permitido reutilizar los buffers especificados en ellas.

El modelo de costes simplificado presentado en la sección anterior sólo puede ser usado cuando el comportamiento del procesador origen esté completamente desacoplado del procesador destino. Es un modelo de comunicaciones que no incorpora mecanismos para tratar con una sincronización artificial entre emisor y receptor del mensaje. De esta forma, sólo va a ser adecuado para las comunicaciones en los modos buffered y estándar (en una arquitectura que no tenga problemas en proporcionar buffers).

Sin embargo, para el caso que estamos caracterizando (comunicaciones punto a punto y colectivas en MPICH-G2, versión 1.2.4) el procesamiento presente en los nodos es despreciable y todos los modos devuelven resultados similares para comunicaciones punto a punto. Además, las comunicaciones colectivas multinivel conscientes de la topología están implementadas a partir de las funciones `MPI_ISend` (no bloqueante, modo estándar) y `MPI_Recv` (bloqueante, modo estándar), por lo que es posible utilizar este modelo para nuestras comunicaciones.

2.4. Comunicaciones colectivas

Las operaciones colectivas MPI permiten iniciar, con una única instrucción, un conjunto de comunicaciones punto a punto para procesar datos contenidos en distintos procesos en una tarea de alto nivel, como son las barreras de sincronización, la difusión de información desde un proceso a todos los demás, la recolección de datos solicitada por un proceso, la distribución de datos hacia todos los procesos o el procesado distribuido de una función para reducir todos los datos a un único valor en un proceso.

En la librería MPICH estas operaciones están implementadas sobre primitivas de operaciones punto a punto. La implementación por defecto, haciendo uso de un comunicador oculto que aísla estas comunicaciones de las de usuario, utiliza para estas funciones un algoritmo binomial por sus características de escalabilidad y un algoritmo hipercubo para las barreras. También existen versiones especiales para algunas arquitecturas, optimizadas por el fabricante.

La librería MPICH-G2 incluye algunas optimizaciones genéricas de bajo nivel y algunas optimizaciones específicas orientadas a las operaciones colectivas. Así, por un lado, ofrece un mejor ancho de banda en la máquina local trabajando de manera más estrecha con la versión MPI del fabricante, si está disponible; consigue una menor latencia en la máquina local, limitando la manera en que consumen los mensajes TCP, e implementa el uso de sockets bidireccionales entre procesos.

Por otro lado, MPICH-G2 ofrece nuevas operaciones multinivel conscientes de la topología a partir de mecanismos automáticos de autodescubrimiento de la misma [140]. Versiones iniciales clasificaban las comunicaciones en *locales al cluster* y *externas*. La versión bajo estudio construye automáticamente toda una jerarquía de niveles de agrupamiento de procesos, en función del protocolo más eficiente, que pasa a estar disponible como un atributo del comunicador. Esta información permite que cada nodo pueda conocer desde qué proceso recibirá un mensaje y a qué otros procesos debe reenviarlo, sin que esto requiera comunicaciones adicionales de control. El objetivo es reducir el número de comunicaciones a través de los protocolos más lentos. La información de topología conocida por cada nodo se corresponde con el esquema mostrado en la tabla 2.1.

El ejemplo anterior representa una ejecución paralela en un sistema de 12 nodos como el que se muestra en la figura 2.4. Todos ellos pueden comunicarse con todos ya que comparten el

rango	0	1	2	3	4	5	6	7	8	9	10	11
profundidad	4	4	4	4	3	3	3	3	3	3	3	3
protocolos:												
wan	0	0	0	0	0	0	0	0	0	0	0	0
lan	0	0	0	0	1	1	1	1	1	1	1	1
local	0	0	0	0	1	1	1	1	2	2	2	2
MPI del fabricante	0	0	0	0								

Tabla 2.1: Información de topología en MPICH-G2 conocida por cada nodo.

mismo identificador en el nivel *wan*. Éste es el nivel más lento. La comunicación es más eficiente para comunicar los 4 primeros entre sí, o los 8 siguientes entre sí (nivel *lan*), porque están ubicados en clusters diferentes. Dentro del primer cluster no hay situaciones privilegiadas. Pero dentro del segundo, las comunicaciones entre los 4 primeros y entre los 4 siguientes son más eficientes por la forma en que los procesos se agrupan en las mismas máquinas (nivel local). Además, los procesos del primer cluster pueden utilizar una implementación especial específica de la arquitectura (nivel MPI del fabricante).

Con esta información, el nodo sabe que enviar directamente información del proceso (rango) 0 al 1 es eficiente, ya que la librería utilizará la implementación propia de MPI. Pero que para enviar información del 0 al 11, lo más eficiente es delegar en el proceso 4, que es el representante del grupo identificado como 1 en el nivel *lan*. A su vez, este proceso delega en el 8, que es el representante del grupo identificado como 2 en el nivel local. Por último, este proceso puede enviar directamente la información al 11 porque están al mismo nivel. Aunque el usuario puede conocer este esquema, no es probable que lo utilice para comunicaciones punto a punto, pero MPICH-G2 sí la utiliza para aumentar automáticamente la eficiencia de las operaciones de comunicación colectivas.

MPICH-G2, versión 1.2.4, es la librería disponible durante el desarrollo del proyecto CrossGrid entre los años 2002 a 2005. Implementa cinco de estas operaciones colectivas multinivel conscientes de la topología: `MPI_Barrier`, `MPI_Bcast`, `MPI_Gather`, `MPI_Scatter` y `MPI_Reduce`. La siguiente versión, 1.2.5.1, se amplía a once operaciones, añadiendo `MPI_Allgather`, `MPI_Allgatherv`, `MPI_Alltoall`, `MPI_Allreduce`, `MPI_Reduce_scatter` y `MPI_Scan`.

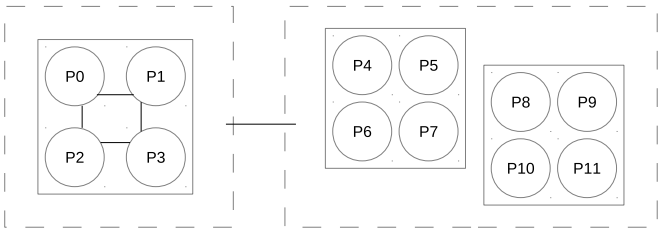

Figura 2.4: Información de topología en MPICH-G2 conocida por cada nodo. Los procesadores P0 a P4 están en la misma máquina, conectados por una red de alta velocidad. Los procesadores P5 a P8 y P9 a P11 están en las mismas máquinas, pero con conexión estándar.

2.4.1. `MPI_Bcast`

Esta función difunde un mensaje desde un proceso inicial, llamado proceso raíz, a todos los procesos del mismo comunicador. Al terminar, los contenidos del buffer de comunicaciones del primero estarán disponibles en el buffer del resto de procesadores.

MPICH-G2 utiliza para la comunicación entre distintos clusters un esquema de comunicaciones *flat tree*: el proceso raíz envía secuencialmente el mensaje a un procesador representante de cada cluster. Es una solución poco sofisticada, pero se espera que en este nivel (*wan*) las comunicaciones sean escasas. En cualquier otro nivel (*lan*, local o la mejor conexión del vendedor) se utiliza un esquema de comunicaciones *binomial tree*, ya sea a un representante de otro grupo o a un proceso. Un árbol binomial B_k de orden $k \geq 0$ es un árbol ordenado de 2^k nodos que se define de manera recursiva: el árbol binomial B_0 consiste en un nodo único; el árbol binomial B_k ($k > 0$) tiene una raíz con k descendientes, donde el descendiente i ($0 < i \leq k$) es la raíz de un árbol binomial B_i (figura 2.5).

Esta división de algoritmos según el nivel de la comunicación se repite en la mayor parte de las nuevas operaciones multinivel colectivas de la librería. En el sistema de 12 nodos de la figura 2.4, `MPI_Bcast` genera el esquema de comunicaciones mostrado en la figura 2.6.

A continuación vamos a definir un modelo para el tiempo de comunicación de la operación `Bcast`, utilizando i como el procesador raíz, a partir de una matriz de coste para un tamaño de mensaje concreto. Se considera que el tiempo de la operación colectiva es el tiempo necesario para que todas las comunicaciones punto a punto generadas por la librería terminen (nótese que estas operaciones colectivas incluyen también una comunicación entre el procesador raíz y él mismo).

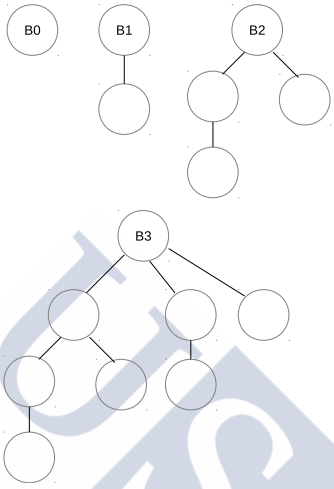

Figura 2.5: Árboles binomiales B_0, B_1, B_2 y B_3. Cada uno contiene a todos los anteriores en una rama.

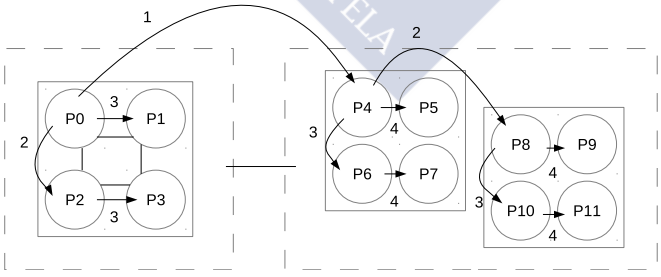

Figura 2.6: Esquema de comunicaciones generado por `MPI_Bcast` en el sistema de la figura anterior. El mensaje con etiqueta 1 se realiza como parte del algoritmo *flat tree* entre clusters, mientras que los demás son de los algoritmos binomiales en su cluster. Las etiquetas indican el ordena en que los mensajes se envían. Dos etiquetas iguales indican que los mensajes son simultáneos.

La topología del Grid impone en las operaciones colectivas un patrón concreto de comunicaciones, que depende tanto del agrupamiento de procesos por protocolos como de la identificación por rango de los procesos dentro de cada grupo. El tiempo de ejecución del algoritmo, designado por T o $T(i)$, porque depende de la elección del nodo raíz, corresponde a la suma de los coeficientes de la matriz de coste t_{ij} para las comunicaciones entre los nodos i y j en ese patrón. El usuario, de manera general, no tiene control sobre qué máquina recibe cada proceso y no puede conocer qué coeficientes de la matriz utilizar para el cálculo. Sin embargo, puede calcular un tiempo máximo y mínimo que acoten el valor real, y analizar en qué porcentaje pueden desviarse los valores reales de los límites fijados. En este sección realizaremos ese análisis.

La primera parte de la operación `Bcast` utiliza un algoritmo *flat tree* para comunicar el primer proceso con un representante de cada uno de los clusters del Grid. En este nivel, el tiempo de comunicación corresponde a la suma secuencial del tiempo empleado en las distintas comunicaciones entre el proceso raíz y cada uno de los representantes. El límite superior para el tiempo del algoritmo *flat tree* desde el proceso i se obtiene cuando la comunicación se realiza con el proceso de mayor coste en cada localización. Para evitar utilizar información de la topología, se puede sustituir el mayor coste de cada cluster por el mayor coste en todo el sistema y repetirlo tantas veces como clusters estén presentes. El límite inferior sigue una expresión similar:

$$\begin{aligned} L_{\text{máx}} &= S \cdot \text{máx}(t_{ij}) \\ L_{\text{mín}} &= S \cdot \text{mín}(t_{ij}) \end{aligned} \tag{2.7}$$

donde S es el número de clusters, y los coeficientes t_{ij} corresponden a las comunicaciones entre el proceso raíz, i, y todos los demás procesos del sistema.

Se puede analizar el error entre el tiempo real del algoritmo, T, y estos límites teóricos calculados. Para ello, se considera el caso ideal donde todas las comunicaciones son iguales, $t_{ij} = t'$. En estas condiciones el límite superior coincide con el valor real:

$$L_{\text{máx}} = T \tag{2.8}$$

y, al mismo tiempo,

$$\begin{aligned} T &= S \cdot t' \\ L_{\text{máx}} &= S \cdot t' \end{aligned} \tag{2.9}$$

Supongamos que se realiza una perturbación mínima en uno de los enlaces, t_{kl}, pasando el valor del coeficiente de t' a $t'+d$, con $d \geq 0$. Podemos distinguir tres casos. El primero supone que el enlace afectado no corresponde a una comunicación desde el nodo raíz a otro nodo, es decir, $k \neq i$. En este caso, el valor real no se ve afectado, pero el límite teórico superior se redefine a partir del nuevo valor:

$$\begin{aligned} T &= S \cdot t' \\ L_{\text{máx}} &= S \cdot (t'+d) \end{aligned} \tag{2.10}$$

Operando, se obtiene la nueva relación entre ellos:

$$L_{\text{máx}} = T \cdot (1 + d/t') \tag{2.11}$$

Esta ecuación implica que cuando $d \to \infty$, $L_{\text{máx}} \to \infty$ y cuando $d \to 0$, $L_{\text{máx}} \to T$. Es decir, para calcular el máximo, utilizar un coeficiente que no intervenga en una comunicación del esquema *flat tree* introduce errores arbitrariamente elevados en el límite.

Sucede algo similar con el límite inferior. Partiendo del caso ideal, si se realiza una perturbación mínima en uno de los enlaces, t_{kl}, pasando el tiempo de t' a $t'-d$, con $d \leq t'$, se encuentra:

$$\begin{aligned} T &= S \cdot t' \\ L_{\text{mín}} &= S \cdot (t'-d) \end{aligned} \tag{2.12}$$

Operando, se obtiene:

$$L_{\text{mín}} = T \cdot (1 - d/t') \tag{2.13}$$

Esta ecuación implica que cuando $d \to t'$, $L_{\text{mín}} \to 0$ y cuando $d \to 0$, $L_{\text{mín}} \to T$. Es decir, para calcular el mínimo, utilizar un coeficiente que no intervenga en la comunicación en el esquema *flat tree* también introduce errores arbitrariamente grandes en el límite inferior (en este caso, tiende a cero).

En resumen, del análisis del primer caso se deduce que la limitación de tanto el límite superior como el inferior deben calcularse sobre los coeficientes de la matriz de coste t_{ij} donde el primer índice corresponda con el proceso raíz. En caso contrario, los valores llevan asociados errores arbitrariamente grandes que no pueden ser identificados de forma precisa.

El segundo caso posible supone que el enlace afectado se corresponde a una comunicación desde el nodo raíz a otro nodo, es decir, $k = i$, pero que el enlace concreto no se utiliza durante el algoritmo *flat tree*. Es inmediato verificar que se obtienen ecuaciones idénticas al caso anterior:

$$\begin{aligned} L_{\text{máx}} &= T \cdot (1 + d/t') \\ L_{\text{mín}} &= T \cdot (1 - d/t') \end{aligned} \tag{2.14}$$

Se deduce que tanto el límite superior como el inferior deben calcularse estrictamente sobre los coeficientes de la matriz de coste t_{ij} donde el primer índice corresponde con el proceso raíz, y el segundo índice corresponde con los nodos que se utilicen en la comunicación *flat tree*. En caso contrario, los valores llevan asociados errores arbitrariamente grandes. En resumen, no se puede ignorar la topología en el caso del cálculo de los límites para este esquema de comunicaciones sin perder precisión.

Por último, podemos considerar la situación en la que una perturbación mínima en un enlace del sistema ideal afecta a un enlace que participa en las comunicaciones. En ese caso, se encuentra para el tiempo y para el límite superior, que:

$$\begin{aligned} T &= S \cdot t' + d \\ L_{\text{máx}} &= S \cdot (t' + d) \end{aligned} \tag{2.15}$$

Operando, se obtiene:

$$L_{\text{máx}} = T \cdot (1 + d \cdot (S-1)/(S \cdot t' + d)) \tag{2.16}$$

Esta ecuación implica que cuando $d \to \infty$, $L_{\text{máx}} \to S \cdot T$ y cuando $d \to 0$, $L_{\text{máx}} \to T$. Es decir, cuando se conocen los coeficientes t_{ij} reales, aquellos que efectivamente participan en la comunicación, y se define el valor teórico máximo en función del mayor de ellos, el error que se comete está acotado:

$$T \leq L_{\text{máx}} = S \cdot \text{máx}(t_{ij}) \leq S \cdot T \tag{2.17}$$

donde los t_{ij} corresponden exclusivamente a los enlaces que participan en el esquema de comunicaciones *flat tree*.

De manera análoga, para el límite inferior:

$$L_{\text{mín}} = T \cdot (1 - d \cdot (S-1)/(S \cdot t' - d)) \tag{2.18}$$

Esta ecuación implica que cuando $d \to t'$, $L_{\text{mín}} \to 0$ y cuando $d \to 0$, $L_{\text{mín}} \to T$. Es decir, cuando se conocen los coeficientes t_{ij} correctos, y se define el valor teórico mínimo en función del menor de ellos, el error que se comete está acotado:

$$0 \leq L_{\text{mín}} = S \cdot min(t_{ij}) \leq T \tag{2.19}$$

donde los t_{ij} corresponden exclusivamente a los enlaces que participan en el esquema de comunicaciones *flat tree*.

En resumen, del análisis del algoritmo *flat tree* se extrae la conclusión de que no es posible modelar con precisión el comportamiento de las comunicaciones en el nivel *wan* sin conocer qué enlaces participan en el mismo. Se puede ignorar esta advertencia y utilizar los valores máximos y mínimos de los coeficientes en la matriz de costes, más sencillos de obtener, para dar límites máximos y mínimos al algoritmo, pero en ese caso se desconocerá el error introducido en el modelo. Sólo cuando se conocen los enlaces y coeficientes t_{ij} que efectivamente participan en las comunicaciones se pueden establecer los errores asociados, usando las ecuaciones anteriores.

La segunda parte de la operación `Bcast` utiliza un algoritmo binomial para copiar la información a los procesos dentro de cada cluster. El nodo que ha recibido la comunicación *flat tree* se convierte en el procesador raíz para este cluster. Sin pérdida de generalidad, vamos a centrar el análisis en uno cualquiera de ellos, donde el nodo i va a comunicarse con todos los demás utilizando un esquema de comunicaciones binomial.

La operación termina cuando todos los procesos reciben el mensaje. El tiempo de comunicación corresponde a la comunicación más costosa en las distintas ramas del árbol binomial, que solapan entre sí. Este tiempo puede ser acotado por un límite superior que corresponde a la rama que acumule las comunicaciones más costosas del cluster.

La primera aproximación para simular la comunicación más costosa es utilizar el mayor coeficiente de coste del cluster repetido tantas veces como enlaces tenga la rama más larga. Para el límite inferior sucede algo similar. A partir del menor coeficiente de coste del cluster, repetido tantas veces como enlaces tenga la rama más corta, se calcula el modelo de la comunicación menos costosa.

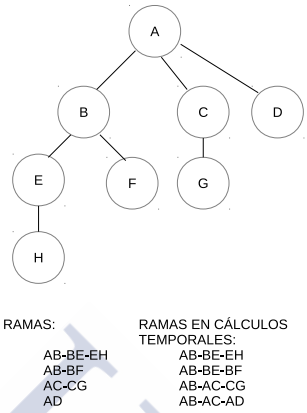

Figura 2.7: Árbol B_3 y los dos modos de definir las ramas. Para los cálculos temporales, las ramas de cada árbol binomial incluyen también los mensajes que se han enviado antes de llegar a ella.

Para analizar el algoritmo se redefine el concepto de las ramas del árbol binomial a nivel de tiempo de comunicación para incluir también las comunicaciones que han tenido que emitirse antes de comenzar cada rama (figura 2.7).

A partir del esquema de comunicaciones binomial, se encuentra que en un árbol binomial de k nodos se realizan comunicaciones encadenadas, donde la comunicación más larga implica $M = \lceil log_2(k) \rceil$ enlaces. Como las ramas del árbol binomial están balanceadas, el camino más corto implicará $N = \lfloor log_2(k) \rfloor$ enlaces. Se vuelve a designar como T el tiempo real del algoritmo binomial, que depende de la elección del nodo raíz.

Las ecuaciones del modelo que se obtienen para el caso binomial son similares a las obtenidas para el caso del esquema *flat tree*:

$$\begin{aligned} L_{\text{máx}} &= M \cdot \text{máx}(t_{ij}) \\ L_{\text{mín}} &= N \cdot \text{mín}(t_{ij}) \end{aligned} \tag{2.20}$$

donde ahora los cálculos corresponden a los coeficientes de la rama que suma el mayor coste y el menor coste, respectivamente. En ese caso, se pueden asignar valores de error a los límites calculados. Y con un desarrollo similar al del caso anterior (todas las comunicaciones en el árbol tienen el mismo valor $t_{ij} = t'$, excepto una, que pasa a tener $t' + d$ o $t' - d$), se obtiene también información sobre la influencia del valor de los coeficientes en las ramas

consideradas:

$$\begin{aligned} L_{\text{máx}} &= T \cdot (1 + d \cdot (M-1)/(M \cdot t' + d))) \\ L_{\text{mín}} &= T \cdot (1 - d \cdot (N-1)/(N \cdot t' - d))) \end{aligned} \tag{2.21}$$

Con lo que se pueden acotar los límites:

$$\begin{aligned} T \leq L_{\text{máx}} = M \cdot \text{máx}(t_{ij}) \leq M \cdot T \\ 0 \leq L_{\text{mín}} = N \cdot \text{mín}(t_{ij}) \leq T \end{aligned} \tag{2.22}$$

obteniendo los coeficientes de la rama que suma el mayor coste y menor coste, respectivamente.

Hay dos diferencias respecto al esquema *flat tree* a la hora de deducir cuál es la rama concreta de mayor coste en un esquema binomial, para así escoger los coeficientes t_{ij} correctos. La primera es que ahora los procesadores pertenecen al mismo cluster y se espera que los enlaces entre ellos sean similares. La segunda es que todas las ramas del árbol se procesan, y todos los coeficientes se utilizan, por lo que se puede establecer una relación entre escoger el $\text{máx}(t_{ij})$ en la rama de mayor coste y escogerlo en el árbol binomial completo. Por simetría, se obtiene que:

$$\text{máx}(t_{ij})^{\text{rama más costosa}} \leq \text{máx}(t_{ij})^{\text{árbol binomial}} \leq M \cdot \text{máx}(t_{ij})^{\text{rama más costosa}} \tag{2.23}$$

Por tanto, podemos utilizar $\text{máx}(t_{ij})^{\text{árbol binomial}}$, que es inmediato de obtener, y seguir teniendo un error acotado. Algo análogo sucede con el límite inferior y $min(t_{ij})^{\text{árbol binomial}}$. Las nuevas ecuaciones del modelo son:

$$\begin{aligned} L_{\text{máx}} &= M \cdot \text{máx}(t_{ij})^{\text{árbol binomial}} \\ L_{\text{mín}} &= N \cdot \text{mín}(t_{ij})^{\text{árbol binomial}} \end{aligned} \tag{2.24}$$

donde ahora t_{ij} recorren el árbol binomial completo. Y sus errores están acotados de la siguiente manera:

$$\begin{aligned} T \leq L_{\text{máx}} = M \cdot \text{máx}(t_{ij})^{\text{árbol binomial}} \leq M^2 \cdot T \\ 0 \leq L_{\text{mín}} = N \cdot \text{mín}(t_{ij})^{\text{árbol binomial}} \leq T \end{aligned} \tag{2.25}$$

Hay un caso particular que se debe considerar. Es posible que dentro de cada cluster existan grupos de procesadores que se comunican con protocolos más eficientes. En ese caso el esquema binomial al nivel analizado sólo va a contactar con el representante del grupo, dejando

las comunicaciones internas al protocolo más eficiente, probablemente también con un esquema de comunicaciones binomial. Eso significa que parámetros como número de procesos y coeficientes de coste de los enlaces de todo el árbol binomial hacen referencia únicamente a los procesos que estén al mismo nivel y a los representantes de cada uno de los grupos, pero no a los procesadores contenidos en el interior de ellos.

Nótese también que, para cada árbol binomial, los valores de M, N y t_{ij} serán diferentes, por lo que se definen tantos límites superiores e inferiores como clusters haya en el Grid.

Resumiendo, hasta el momento hemos obtenido valores máximos y mínimos para caracterizar la comunicación *flat tree* en el nivel más externo de la operación colectiva, así como valores máximos y mínimos para la comunicación *binomial tree* en cada cluster. Sus resultados son:

$$\begin{aligned} L_{\text{máx}}^{\text{flat}} &= S \cdot \text{máx}(t_{ij}) \\ L_{\text{mín}}^{\text{flat}} &= S \cdot \text{mín}(t_{ij}) \end{aligned} \qquad (2.26)$$

$$\begin{aligned} L_{\text{máx}}^{\text{binomial}} &= M \cdot \text{máx}(t_{ij}) \\ L_{\text{mín}}^{\text{binomial}} &= N \cdot \text{mín}(t_{ij}) \end{aligned}$$

donde los coeficientes t_{ij} de las ecuaciones *flat tree* hacen referencia a los enlaces que efectivamente se usan en este nivel, mientras que los t_{ij} de las ecuaciones binomiales hacen referencia a todos los enlaces internos del árbol binomial en el cluster. Bajo estas condiciones se conocen los errores aceptados de los valores máximos y mínimos:

$$\begin{aligned} T^{\text{flat}} \leq L_{\text{máx}}^{\text{flat}} &\leq S \cdot T^{\text{flat}} \\ 0 \leq L_{\text{mín}}^{\text{flat}} &\leq T^{\text{flat}} \end{aligned} \qquad (2.27)$$

$$\begin{aligned} T^{\text{binomial}} \leq L_{\text{máx}}^{\text{binomial}} &\leq M^2 \cdot T^{\text{binomial}} \\ 0 \leq L_{\text{mín}}^{\text{binomial}} &\leq T^{\text{binomial}} \end{aligned}$$

A continuación estableceremos las ecuaciones para la operación completa a partir de estos límites superiores e inferiores.

El tiempo total de la operación debe tener en cuenta tanto el tiempo calculado para el algoritmo *flat tree* como el calculado para el algoritmo *binomial tree* dentro de cada cluster, atendiendo

a que las comunicaciones dentro de cada uno no comienzan hasta haber sido seleccionados por el algoritmo de primer nivel. El límite superior global se construye de manera inmediata como la combinación secuencial del límite superior del esquema *flat tree* y del binomial. De manera general, estas comunicaciones pueden solaparse en el tiempo, por lo que considerarlas consecutivas establecen un límite superior realista:

$$L_{\text{máx}}^{\text{bcast}} = L_{\text{máx}}^{\text{flat}} + \text{máx}(L_{\text{máx}}^{\text{binomial}}) \tag{2.28}$$

Si se cumplen las condiciones de selección de coeficientes, estos tiempos están acotadas por:

$$T^{\text{flat}} + T^{\text{binomial}} \leq L_{\text{máx}}^{\text{bcast}} \leq S \cdot T^{\text{flat}} + (M^{mc})^2 \cdot T^{\text{binomial}} \tag{2.29}$$

donde S es el número de clusters, mc es el cluster con el mayor $L_{\text{máx}}^{\text{binomial}}$, M^{mc} es la parte entera por exceso del logaritmo en base 2 del número de nodos del cluster mc, y T^{flat} y T^{binomial} son, respectivamente, el valor real del tiempo de comunicación en el *flat tree* y el valor real del tiempo de comunicación en el *binomial tree* en ese cluster.

De manera análoga, se puede definir un límite inferior global como la combinación paralela del límite inferior del esquema *flat tree* y del binomial:

$$L_{\text{mín}}^{\text{bcast}} = \text{mín}(L_{\text{mín}}^{\text{flat}}, \text{mín}(L_{\text{mín}}^{\text{binomial}})) \tag{2.30}$$

De nuevo, con las condiciones impuestas sobre los coeficientes de cada término, estos valores están acotados por:

$$0 \leq L_{\text{mín}}^{\text{bcast}} \leq \text{mín}(T^{\text{flat}}, T^{\text{binomial}}) \tag{2.31}$$

En resumen, el tiempo de ejecución de la operación colectiva `Bcast` en un Grid genérico está acotado entre $L_{\text{máx}}^{\text{bcast}}$ y $L_{\text{mín}}^{\text{bcast}}$, que pueden ser calculados a partir de la matriz de costes de comunicación. Estos costes dependen de características físicas de los enlaces, de la topología del Grid y del tamaño del mensaje. Y, bajo ciertas condiciones, los límites globales, pueden ser acotados para proporcionar una medida del error de la aproximación.

El principal problema de esta aproximación es conocer a priori cuáles son los coeficientes t_{ij} involucrados en las comunicaciones entre clusters y dentro de los clusters. Las asociaciones entre procesos y nodos, de manera general, no están bajo el control del usuario. De manera que hay que introducir conocimiento de la topología en el modelo para obtener estos valores.

2.4.2. `MPI_Scatter`

En una operación `MPI_Scatter`, el proceso raíz envía el contenido de su buffer de comunicaciones, dividido en n fragmentos, a los procesos del grupo, de tal manera que el proceso de rango i recibe sólo el i-ésimo fragmento. Conceptualmente equivale a enviar desde el proceso raíz un mensaje a cada proceso, ordenados por rango, con sólo un fragmento del mensaje total.

La implementación utiliza un esquema de comunicaciones combinación de *flat tree* y *binomial tree* como en el caso del `Bcast`. Sin embargo, el tamaño de las comunicaciones no es constante sino que depende del nivel en que esté situado el nodo destino. Cada comunicación tiene que llevar los datos pertenecientes al nodo destino y a todo el árbol de comunicaciones que parte de ese. En el nivel superior, con el algoritmo *flat tree*, cada comunicación tiene un tamaño proporcional al número de nodos de cada cluster. En los siguientes niveles, en cada paso discreto del algoritmo *binomial tree*, cada comunicación reduce su tamaño a la mitad.

En el modelo de la matriz de costes, cada valor t_{ij} depende del tamaño del mensaje. Siendo el menor mensaje de tamaño w, que consideraremos como tamaño base, el mayor mensaje tendrá un tamaño $N^m \cdot w$, donde N^m es el número de nodos en el cluster con mayor número de nodos (cluster m). Este mensaje sucede en el nivel del algoritmo *flat tree*. Alternativamente, para ocultar detalles de la topología, su puede sustituir el número de nodos del cluster, N^m, por el número total de nodos del Grid, N.

Hacemos dos suposiciones. Primera, que los valores de la matriz de coste aumentan al aumentar el tamaño del mensaje. Segunda, que el coeficiente t_{ij} para el mensaje de tamaño $N \cdot w$ es aproximadamente igual a $N \cdot t_{ij}$, donde t_{ij} corresponde a la matriz de costes para el mensaje w. En ese caso, se puede utilizar esta nueva matriz de coste, con el tamaño de mensaje ampliado, para calcular el límite superior con las mismas ecuaciones que en el caso del `Bcast`. Para el límite inferior se puede utilizar la misma expresión que en el caso del `Bcast` utilizando la matriz de coste con el tamaño de mensaje base, es decir:

$$\begin{aligned} L^{\text{scatter}}_{\text{máx}}(\text{mensaje de tamaño } w) &= L^{\text{bcast}}_{\text{máx}}(\text{mensaje de tamaño } w \cdot N) \\ &= N \cdot L^{\text{bcast}}_{\text{máx}}(\text{mensaje de tamaño } w) \\ L^{\text{scatter}}_{\text{mín}}(\text{mensaje de tamaño } w) &= L^{\text{bcast}}_{\text{mín}}(\text{mensaje de tamaño } w \cdot N) \\ &= N \cdot L^{\text{bcast}}_{\text{mín}}(\text{mensaje de tamaño } w) \end{aligned} \tag{2.32}$$

2.4.3. MPI_Gather

Es la operación recíproca a `MPI_Scatter`. Cada proceso del grupo, incluido el proceso raíz, envía el contenido de su buffer de comunicaciones al proceso raíz. Éste recibe los mensajes y los almacena ordenados por el rango del emisor. El resultado es conceptualmente equivalente a que todos los procesos, ordenados por rango, envíen un mensaje al proceso raíz que los almacena de forma consecutiva para obtener el conjunto de todos los valores.

Utiliza un esquema de comunicaciones combinación de *flat tree* y *binomial tree* como en el caso del `Bcast`, recorriéndose ahora de abajo hacia arriba. De nuevo el tamaño de las comunicaciones no es constante, sino que depende del nivel en que esté situado el nodo origen. Cada comunicación tiene que enviar los datos pertenecientes a su nodo y a todo el árbol de comunicaciones que parte de ese. En el nivel superior, con el algoritmo *flat tree*, cada comunicación tiene los mensajes más grandes, con un tamaño que es, de nuevo, proporcional al número de nodos de cada cluster. Las ecuaciones coinciden con las del caso anterior:

$$\begin{aligned} L^{\text{gather}}_{\text{máx}}(\text{mensaje de tamaño } w) &= L^{\text{bcast}}_{\text{máx}}(\text{mensaje de tamaño } w \cdot N) \\ &= N \cdot L^{\text{bcast}}_{\text{máx}}(\text{mensaje de tamaño } w) \\ L^{\text{gather}}_{\text{mín}}(\text{mensaje de tamaño } w) &= L^{\text{bcast}}_{\text{mín}}(\text{mensaje de tamaño } w \cdot N) \\ &= N \cdot L^{\text{bcast}}_{\text{mín}}(\text{mensaje de tamaño } w) \end{aligned} \tag{2.33}$$

2.4.4. MPI_Reduce

La operación `MPI_Reduce` combina los elementos proporcionados en el buffer de comunicaciones de cada proceso del grupo, utilizando una operación de reducción, y devuelve el valor combinado al proceso raíz. La operación de reducción puede ser cualquiera de una lista predefinida de operaciones (máximo, mínimo, suma, producto, `AND` lógico, `AND` binario, `OR` lógico, `OR` binario, `XOR` lógico, `XOR` binario, valor máximo con el índice de cluster o valor mínimo con el índice de cluster) o estar definida por el usuario. La operación debe ser asociativa. El esquema de comunicaciones resultante de esta operación depende de si la función solicitada es o no conmutativa.

Para funciones conmutativas, se utiliza un esquema de comunicaciones combinación de *flat tree* y *binomial tree* como en el caso del `Bcast`, que se recorre desde abajo hacia arriba. El

tamaño de las comunicaciones es constante. Las ecuaciones coinciden con las del `Bcast`:

$$\begin{aligned} L_{\text{máx}}^{\text{reduce}}(\text{mensaje de tamaño } w) &= L_{\text{máx}}^{\text{bcast}}(\text{mensaje de tamaño } w) \\ L_{\text{mín}}^{\text{reduce}}(\text{mensaje de tamaño } w) &= L_{\text{mín}}^{\text{bcast}}(\text{mensaje de tamaño } w) \end{aligned} \tag{2.34}$$

Para funciones no conmutativas, no se pueden utilizar estos esquemas de comunicaciones porque la distribución de procesos a lo largo del Grid impone un orden en el cálculo de la operación. Para asegurar un orden y resultado correctos, el cálculo distribuido de la reducción se sustituye por una operación de `Gather` hacia el proceso raíz y una reducción secuencial en el mismo. Las ecuaciones coinciden con las del `Gather`. Se ignora conscientemente el tiempo de cálculo de la función de reducción en el nodo raíz.

$$\begin{aligned} L_{\text{máx}}^{\text{reduce}}(\text{mensaje de tamaño } w) &= L_{\text{máx}}^{\text{bcast}}(\text{mensaje de tamaño } N \cdot w) \\ L_{\text{mín}}^{\text{reduce}}(\text{mensaje de tamaño } w) &= L_{\text{mín}}^{\text{bcast}}(\text{mensaje de tamaño } N \cdot w) \end{aligned} \tag{2.35}$$

La versión 1.2.4 de MPICH-G2 muestra una particularidad en la implementación de las operaciones `MPI_Scatter` y `MPI_Gather`. En los mensajes punto a punto generados como consecuencia de la operación colectiva se envían el tipo de datos especial `MPI_PACKED` en lugar de los datos `MPI_INT`, `MPI_LONG`, `MPI_FLOAT`, `MPI_DOUBLE`, `MPI_CHAR` o similares, que tiene el mismo tamaño de los datos que empaqueta más un byte de control, de manera que el tamaño de mensaje aumenta en un byte respecto al esperado. Debido al intercambio de mensajes de este tipo de datos, aparece, para cada nodo, un nuevo mensaje al terminar las comunicaciones *flat tree* y *binomial tree* que sirve para convertir entre el tipo empaquetado y el tipo de datos nativo. Esta comunicación es interna al nodo, se implementa utilizando la operación `SendRecv`, pero no pasa a la red. Puesto que la función `MPI_Reduce` reutiliza las implementaciones de `MPI_Bcast` y `MPI_Gather`, también aparecen comunicaciones con el tipo de datos nativo para funciones de reducción conmutativas y con el tipo de datos `MPI_PACKED` para funciones de reducción no conmutativas.

2.4.5. `MPI_Barrier`

Esta función bloquea el proceso hasta que todos los miembros del grupo la hayan ejecutado, momento en el que termina. De esta manera se obtiene una barrera de sincronización entre todos los procesos.

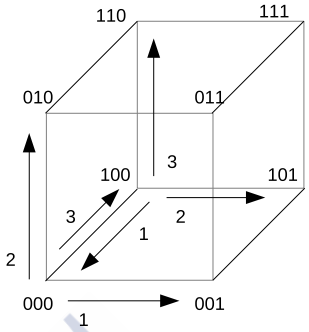

Figura 2.8: Algoritmo de comunicación en hipercubo. Los nodos se codifican mediante una secuencia binaria de bits, de manera que los nodos conectados difieren en un único dígito. En la figura aparecen las primeras comunicaciones (señal `esperar`) de los nodos 000 y 100 a sus vecinos.

Utiliza un algoritmo *flat tree* para sincronizar los procesos entre los distintos clusters del Grid y un algoritmo de comunicación en hipercubo para sincronizar los procesos dentro de cada cluster. El algoritmo *binomial tree* no se considera adecuado por tener una latencia variable.

En el nivel superior cada proceso del conjunto, un proceso representativo por cada cluster, envía un mensaje vacío, funcionando como señal `esperar`, a todos los otros procesos y recibe también un mensaje vacío del resto de procesos, mensaje que funciona como señal `continuar`, como se ve en la figura 2.8. Nótese que en el modelo de comunicaciones de matriz de costes, el coste del mensaje vacío no es nulo. En las operaciones colectivas anteriores se ha asumido que este coste es proporcional al tamaño del mensaje, lo que es una aproximación válida para grandes tamaños de mensaje. La matriz de coste que estamos utilizando ya tiene en consideración un valor no nulo para el tamaño de mensaje nulo.

En el nivel superior cada nodo envía S mensajes consecutivos y recibe S mensajes consecutivos, siendo S el número de clusters presentes en el Grid. En total se realizan $2 \cdot S$ pasos discretos. Las ecuaciones son:

$$\begin{aligned} L_{\text{máx}}^{\text{flat}} &= 2S \cdot \text{máx}(t_{ij}) \\ L_{\text{mín}}^{\text{flat}} &= 2S \cdot \text{mín}(t_{ij}) \end{aligned} \tag{2.36}$$

donde t_{ij} son las comunicaciones entre un proceso representativo por cada cluster y los procesos de las otras.

En los niveles inferiores, cada proceso envía un mensaje vacío a sus $log_2(k)$ vecinos en el hipercubo definido en el cluster, siendo k el número de procesos en el cluster, y espera un mensaje vacío de sus $log_2(k)$ vecinos en el hipercubo. En caso de que el hipercubo esté incompleto porque no haya procesadores disponibles suficientes, se simulan los procesos virtuales necesarios pero que no reciben ni generan mensajes. Cada nodo de cada cluster envía K mensajes consecutivos y recibe K mensajes consecutivos, siendo $K = log_2(k)$. En total son $2 \cdot K$ pasos discretos. Las ecuaciones son:

$$\begin{aligned} L_{\text{máx}}^{\text{hipercubo}} &= 2K \cdot \text{máx}(t_{ij}) \\ L_{\text{mín}}^{\text{hipercubo}} &= 2K \cdot \text{mín}(t_{ij}) \end{aligned} \tag{2.37}$$

El proceso total tendrá unos límites superior e inferior que combinen los de ambos algoritmos:

$$\begin{aligned} L_{\text{máx}}^{\text{barrier}} &= L_{\text{máx}}^{\text{flat}} + \text{máx}(L_{\text{máx}}^{\text{hipercubo}}) \\ L_{\text{mín}}^{\text{barrier}} &= \text{mín}(L_{\text{mín}}^{\text{flat}}, \text{mín}(L_{\text{mín}}^{\text{hipercubo}})) \end{aligned} \tag{2.38}$$

La versión 1.2.4 de MPICH-G2 muestra una particularidad en la implementación de la función `MPI_Barrier`. La librería lanza una barrera al terminar el programa en caso de que lo considere necesario, por ejemplo, cuando se han realizado operaciones colectivas en algún momento pero no se ha forzado una barrera posterior.

USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA

CAPÍTULO 3

MÉTODOS ITERATIVOS EN ARQUITECTURAS DISTRIBUIDAS

Este capítulo analiza y modeliza una librería de álgebra matricial dispersa que hace uso simultáneamente de paralelismo de datos y paralelismo por pase de mensajes. En primer lugar, se muestran someramente las principales técnicas de resolución iterativa de sistemas lineales de ecuaciones. A continuación, se discuten las características de la librería PARAISO. Finalmente, se caracterizan las computaciones y comunicaciones de la librería para definir un modelo de coste de los diferentes algoritmos que la componen.

3.1. Métodos iterativos

Muchas aplicaciones de cálculo científico necesitan la resolución de grandes sistemas de ecuaciones lineales. Su forma básica es una ecuación matricial $A \cdot x = b$, donde, a partir de A y b, se resuelve x. La estructura de la matriz A depende de la aplicación [86][17][229].

Los sistemas con una matriz de coeficientes densa pueden ser atacados por métodos directos, basados en factorizaciones. Pero cuando las matrices de coeficientes tienen una forma característica y un elevado número de entradas nulas, formando lo que se denominan matrices dispersas, se utilizan otro tipo de técnicas que, como son los métodos iterativos, que se adaptan mejor a estos patrones y que resultan más eficientes en términos de tiempo y restricciones de memoria [86][17].

Los métodos iterativos [28] son un amplio conjunto de técnicas que usan aproximaciones sucesivas para obtener una solución más precisa de un sistema lineal en cada etapa, bien a través de la relajación de coordenadas o basándose en procesos de proyección que representan una forma canónica de un subespacio para la extracción de la solución aproximada. Los primeros son los métodos estacionarios. Basados en la relajación de coordenadas, se comienza con una solución aproximada y se modifican los componentes de la aproximación hasta que alcanza la convergencia. Las iteraciones pueden expresarse como: $X_k = A \cdot x_{k-1} + b$. Por el contrario, los métodos no estacionarios están basados en la idea de secuencias ortogonales de vectores. Los más populares pertenecen al denominado conjunto de los métodos del subespacio de Krylov. Dado un sistema lineal $A \cdot x = b$, un vector solución inicial x_0 y un vector residuo inicial $r_0 = b - A \cdot x_0$, se define el subespacio de Krylov $K^i(A, r_0)$ de dimensión i como el subespacio cubierto por los vectores $r_0, A \cdot r_0, A^2 \cdot r_0 ... A^{i-1} \cdot r_0$.

Dependiendo de las características de la matriz que define el problema es preferible aplicar unos métodos u otros. La elección del método iterativo adecuado para un problema representado por un sistema lineal concreto es de especial importancia para obtener un buen rendimiento, o incluso para alcanzar la convergencia.

El mejor método a aplicar depende del patrón y características de la matriz de coeficientes, por lo que esta información debe estar disponible para el usuario. Un posible esquema para la selección del método iterativo se ofrece en la bibliografía [28][186]. Muchas veces, es posible modificar la matriz de coeficientes buscando unas características más favorables al método iterativo, lo que da lugar a una mejora del rendimiento al reducir el número de iteraciones para obtener la solución. Un precondicionador es una transformación del sistema lineal original en otro con la misma solución, pero más rápido de resolver con un método iterativo. Usando precondicionadores se mejora la convergencia y la eficiencia de los métodos iterativos, aunque, en general, su eficacia depende de la elección particular de precondicionador y método. La reducción en el número de iteraciones debe compensar el coste de creación y aplicación del precondicionador.

En cualquier método iterativo se encuentran varios núcleos computacionalmente costosos: productos escalares, actualizaciones de vector, producto matriz dispersa-vector o cálculo del precondicionador. Entre ellos, el producto matriz dispersa-vector consume la mayor parte de tiempo de cálculo en cada una de las iteraciones del método, lo que exige una implementación eficiente.

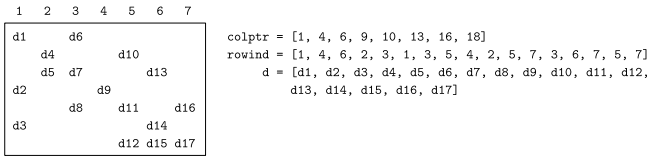

Figura 3.1: Formato de matriz dispersa comprimido por columnas, CSC.

Las matrices dispersas son estructuras de datos irregulares. En ellas, la mayoría de los elementos son nulos. Para sistemas lineales grandes, no es posible almacenarlos como arrays, sino que deben ser representadas utilizando formatos especiales, comprimidos, para aprovechar los recursos del sistema [28]. Esto genera un patrón de accesos irregulares que hace más difícil implementar algoritmos eficientes. Se almacenan únicamente los elementos no nulos y las posiciones que ocupan. Existen diversos esquemas de almacenamiento para matrices dispersas, como *Compressed Sparse Column* (CSC) y *Compressed Sparse Row* (CSR).

El formato CSC, por ejemplo, utiliza tres vectores para describir la matriz. El primero (d) contiene los valores de las entradas no nulas de la matriz ordenadas por columnas. El segundo, el índice de la columna de esas mismas entradas (*rowind*). Y el tercero, indica el comienzo de cada fila ($colptr$). Un ejemplo simple de este almacenamiento se muestra en la figura 3.1.

Los métodos iterativos son populares en implementaciones paralelas, y existen diversas propuestas disponibles como librerías. En este capítulo examinaremos la implementación de una de estas librerías para extraer un modelo de su rendimiento.

3.2. PARAISO

La resolución de sistemas lineales de ecuaciones con un modelo de programación de paralelismo de datos, por ejemplo en *High Performance Fortran* (HPF), es poco eficiente para matrices dispersas irregulares. El problema principal reside en la dificultad de adaptar los accesos irregulares propios de estas operaciones matriciales a los modelos de datos disponibles en el lenguaje. El uso de formatos comprimidos para almacenar las matrices dispersas dificulta aún más el análisis de las distribuciones de datos, e impiden que los compiladores reconozcan el paralelismo existente y eliminen sincronizaciones y comunicaciones redundantes. En la bibliografía se ofrecen técnicas para resolver estos problemas con nuevas directivas de distribución de datos y funciones intrínsecas [72][76]. En las especificaciones de HPF-2 se

incluyeron distribuciones y directivas que pueden facilitar la implementación de problemas irregulares, pero que no estaban disponibles en el momento del desarrollo de esta librería.

PARAISO (*Parallel Iterative Solver*) [186] es una librería de métodos iterativos desarrollada en el Departamento de Electrónica y Computación de la USC. Implementa nuevas técnicas para explotar el paralelismo de los métodos iterativos aplicados a sistemas lineales dispersos usando funciones intrínsecas de HPF en los núcleos computaciones críticos: productos escalares, actualizaciones vectoriales, producto matriz dispersa-vector y precondicionadores [34][36]. Una segunda versión de la librería se ha desarrollado añadiendo el modelo de programación de pase de mensajes para ciertas operaciones críticas [42][35]. Originalmente fue diseñada para una arquitectura concreta, pero el uso de HPF y MPI facilita su portabilidad a otros sistemas distribuidos.

La librería incluye los métodos iterativos de Gradiente conjugado (CG), Gradiente biconjugado (BiCG), Gradiente biconjugado estabilizado (BiCGStab), Gradiente conjugado cuadrático (CGS), Mínimo residuo generalizado (GMRES), Jacobi, Residuo casi mínimo (QMR) y Sobrerrelajación sucesiva (SOR). Incorpora también los precondicionadores Jacobi, Sobrerrelajación sucesiva simétrica (SSOR), Factorización Incompleta (ILU(0)), Factorización Incompleta con Umbral ILUT, precondicionador polinomial de Neumann y precondicionador polinomial por mínimos cuadrados.

El modelo de programación de paralelismo de datos requiere que la distribución de los datos en las memorias locales ofrezca un buen balanceo para ser eficiente. La matriz dispersa de coeficientes es almacenada en un formato apropiado, por ejemplo en CSC, que utiliza tres vectores unidimensionales, y el vector de términos independientes como un array unidimensional.

Los métodos iterativos no estacionarios realizan en cada iteración operaciones de producto escalar, actualización de vectores y productos matriz-vector. Como los vectores y la matriz están distribuidos entre los procesadores, son necesarias comunicaciones en cada iteración. Una implementación eficiente tratará de minimizarlas mediante la alineación apropiada de vectores, uso de funciones propias del lenguaje, y distribuciones de datos permitidas por HPF.

En este capítulo se caracterizan los códigos irregulares MPI de PARAISO antes de su ejecución, desarrollando modelos para la predicción del rendimiento. Esto proporciona información valiosa acerca de aspectos teóricos y prácticos de los métodos iterativos que pueden ayudar al

usuario a comprender y optimizar la ejecución de estos métodos sobre problemas concretos. El objetivo es caracterizar el coste de la resolución de sistemas lineales por métodos iterativos a partir información específica del problema, como el tamaño o el patrón de la matriz de coeficientes que lo describen. Si además incorporamos modelos analíticos simples de la arquitectura de la máquina, es posible estimar el coste de las computaciones y comunicaciones, y, así, realizar una estimación del tiempo de computación de cada iteración del método.

3.3. Caracterización de las computaciones

La librería PARAISO fue desarrollada para una arquitectura concreta que incluía herramientas de análisis de datos de traza sobre código instrumentalizado en forma de número y volumen de mensajes de cada línea. El análisis se extendió con métricas propias, instrumentalizando el código y las librerías del sistema, para obtener información precisa sobre qué procesador envía y recibe cada mensaje. Con todo ello, se obtiene un resumen completo de las computaciones y comunicaciones, más una visión detallada de los eventos de comunicación, que permiten reconstruir el patrón de comunicaciones para los distintos núcleos computacionales.

Entre los núcleos más costosos computacionalmente dentro de la librería PARAISO encontramos el producto matriz dispersa vector, que aparece dentro de cada iteración de todos los métodos disponibles en la librería. Su implementación debe ser lo más eficiente posible porque determina el rendimiento global del método. Éste es uno de los núcleos que se han convertido a MPI para su optimización. Otro de los núcleos computacionales costosos es la resolución triangular dispersa, que se utiliza en ciertos precondicionadores como las factorizaciones incompletas o el SSOR. Estos métodos reutilizan las mismas rutinas que en el caso producto matriz dispersa vector, y necesitan una etapa previa de preprocesamiento para la formación del precondicionador, para luego aplicar dicho precondicionador en cada iteración.

3.3.1. Producto matriz dispersa-vector en HPF

El producto matriz dispersa-vector tiene un coste crítico en las funciones de PARAISO. La librería implementa este núcleo para matrices almacenadas en formatos CSC y CSR. En estos códigos, los vectores x e y del problema están alineados y distribuidos en forma BLOCK, para aumentar la eficiencia de las actualizaciones de vector y los productos escalares. Los lazos del producto matricial son paralelizables, pero las indirecciones y el desalineamiento de los vectores de la matriz dispersa impiden detectarlo automáticamente [72]. Se ha podido reorga-

```
1  INTEGER, DIMENSION(N+1)      :: colptr
2  INTEGER, DIMENSION(Z)        :: rowind
3  REAL, DIMENSION(Z)           :: d
4  REAL, DIMENSION(N)           :: x
5  REAL, DIMENSION(N)           :: y
6  REAL, DIMENSION(Z)           :: aux
7  LOGICAL, DIMENSION(Z)        :: segment
8
9  !HPF$ ALIGN (:) WITH x(:)    :: y
10 !HPF$ ALIGN (:) WITH d(:)    :: rowind, aux, segment
11 !HPF$ DISTRIBUTE (BLOCK)     :: d, x
12 !HPF$ DISTRIBUTE (*)         :: colptr
13
14 y = ZERO
15 aux(colptr(:N)) = x
16 aux = COPY_PREFIX(aux, SEGMENT = segment)
17 aux = d * aux
18 y = SUM_SCATTER(aux, y, rowind)
```

Figura 3.2: Código optimizado del producto matriz dispersa-vector en HPF para la librería PARAISO.

nizar el código, apoyándose en vectores auxiliares para reemplazar estos lazos por llamadas a procedimientos intrínsecos y de librería, de manera que se obtiene una implementación efectivamente paralela [76][224][197]. El código aparece en la figura 3.2. Los detalles pueden consultarse en la bibliografía [36].

La segunda versión de la librería PARAISO sustituye núcleos básicos basados en operaciones intrínsecas por una implementación basada en programación por pase de mensajes, utilizando MPI, donde se programan explícitamente los intercambios de datos y cálculos. De esta manera, las operaciones que involucran vectores, como actualizaciones o productos, se mantienen, pero las operaciones con la matriz se reimplementan. Esto proporciona mayor control sobre los datos y permite obtener un buen balanceo de la carga de computación.

El principal problema para implementar el nuevo núcleo del producto matriz-dispersa vector es la distribución de los datos entre las memorias locales. Las distribuciones disponibles en HPF no son adecuadas para operaciones con la matriz dispersa. Así que resulta necesario hacer redistribuciones de datos desde la distribución BLOCK utilizada por los vectores provenientes de los núcleos HPF, a una que sea eficiente para la matriz en MPI, distribución cíclica, y viceversa. Este intercambio origina altos volúmenes de comunicaciones. Se debe escoger una

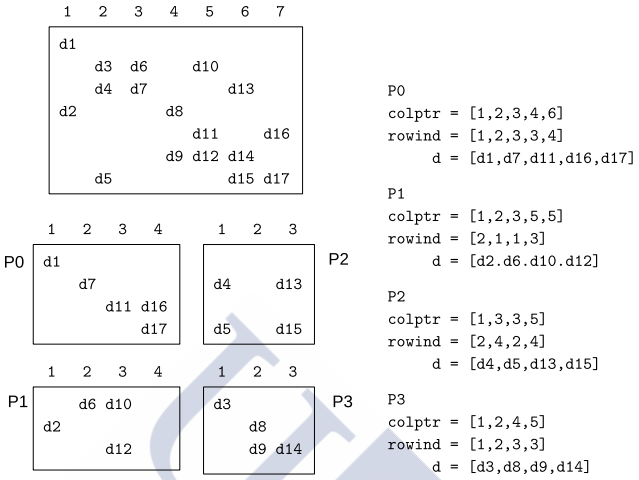

Figura 3.3: Distribución BSC de una matriz dispersa en una red de 2×2 procesadores.

nueva distribución más apropiada para explotar el pase de mensajes con buen balanceo de la carga, y se ha comprobado que BCS (*Block Column Scatter*) obtiene buenos resultados para los métodos de esta librería [194].

La distribución BCS utiliza una proyección cíclica de la matriz sobre una malla de $P \times Q$ procesadores. La matriz se divide de acuerdo a una plantilla $P \times Q$ y cada procesador toma los valores no nulos que coinciden con su posición en la plantilla, y los almacena localmente en un esquema CSC, con tres vectores, tal como se ejemplifica en la figura 3.3.

3.3.2. Producto matriz dispersa-vector en MPI

Como se ha comentado, para la ejecución eficiente del producto matriz dispersa-vector, teniendo en cuenta la distribución BCS de la matriz, es necesario, en cada iteración del método, redistribuir el vector de entrada x, desde una distribución por bloques a otra cíclica por columnas. A continuación se realiza el producto, obteniendo un vector con una distribución cíclica por filas. Finalmente, se redistribuye a bloques para obtener el vector de salida y. En la figura 3.4 se muestra este proceso.

Las operaciones para la primera redistribución son las siguientes:

```
FOR i in iteraciones:
    operaciones HPF
    y = producto_matriz_dispersa_vector_mpi(A, x)
    operaciones HPF

producto_matriz_dispersa_vector_mpi(A, x):
    xcc = redistribuir_block2cyclic_cols(x)
    ycr = producto_matriz_dispersa_vector(A, xcc)
    y = redistribuir_cyclic_rows2block(ycr)
    return y
```

Figura 3.4: Esquema del producto matriz dispersa-vector en MPI para la librería PARAISO.

- etapa de preprocesamiento, donde se calcula una única vez, y sin comunicaciones, qué datos van a ser enviados a cada procesador y con qué desplazamiento (*stride*).
- comunicaciones por filas con una operación colectiva MPI
- comunicaciones por columnas con una operación colectiva MPI, para completar los datos en cada procesador

En la segunda redistribución, dada la redundancia de datos, sólo es necesaria una etapa de comunicaciones por columnas.

El modelo de rendimiento propuesto determina el número de FLOPs de cada núcleo computacional de PARAISO en función de parámetros del problema y, basándose en ese dato, modela el número de FLOPs para una iteración de cada método. Debemos diferenciar entre el número de FLOPs para la primera iteración y el número de FLOPs para cada una de las restantes iteraciones, ya que la primero iteración suele ser más costosa por las inicializaciones y el cálculo del criterio de parada. A pesar de que no poder predecir a priori el número de iteraciones necesarias para la convergencia, se puede comparar el coste computacional relativo de los métodos a partir del coste por iteración. En este caso, los FLOPs son una medida adecuada, porque los métodos iterativos en PARAISO dependen fundamentalmente de computaciones en punto flotante.

En el análisis se incluye también el número de FLOPs por cada precondicionador. En estos casos se consideran los costes de construcción y de aplicación del precondicionador en la ite-

núcleo	FLOPs
spmatvec:	$2a$
spmatvectrans:	$2a-1$
dotproduct:	$2n-1$
jacobi_split:	n
sor_split:	$a+n$
stoptest:	$2n$
norm_inf:	a
triang:	$2a-n$

Tabla 3.1: Núcleos básicos. Modelo de FLOPs para núcleos básicos, siendo n la dimensión de la matriz y a el número de entradas no nulas de la matriz. El núcleo *spmatvec* corresponde al producto matriz dispersa-vector, *spmatvectrans* a su traspuesta y *dotproduct* al producto escalar.

ración. Pero son sólo estimaciones ya que pueden ser necesarios parámetros solo disponibles en tiempo de ejecución para modelar el funcionamiento de algunos precondicionadores.

Tanto en precondicionadores como en núcleos computacionales, la estimación obtenida ofrece expresiones algebraicas sencillas que tienen en cuenta parámetros de la matriz dispersa, como el número de elementos no nulos (a) o su dimensión (n), y cierta información sobre las operaciones, como el grado del polinomio en algunos precondicionadores (g). En aquellos casos donde la estimación depende de parámetros dinámicos, como el patrón de la matriz de coeficientes, se asumen valores por defecto para estos parámetros para realizar la estimación sobre un caso representativo.

A partir de los valores obtenidos utilizando contadores hardware en la ejecución de los diferentes núcleos computacionales, métodos y precondicionadores, hemos obtenido las ecuaciones básicas simplificadas para operaciones en punto flotante. Un análisis detallado de cómo derivar estas ecuaciones se puede consultar en [186][35][37][39][40][38]. Los resultados se muestran en las tablas 3.1, 3.2 y 3.3.

3.4. Caracterización de las comunicaciones

El esquema de comunicaciones generado por estos métodos es fundamental para predecir su rendimiento. La mayor parte de las herramientas para análisis automático se enfocan en el protocolo de pase de mensajes, haciendo una recolección de datos en tiempo de ejecución para posteriormente analizar las trazas y evaluar el rendimiento. El proceso implica instrumen-

método	FLOPs (1a iter)	FLOPs (resto de iters)
CG:	$19n+7a-3$	$12n+2a$
BiCG:	$23n+9a-4$	$16n+4a-1$
BiCGStab:	$29n+9a$	$22n+4a+3$
CGS:	$25n+9a-4$	$18n+5a-1$
GMRES:	$(r^2+6r+13)n+(2r+7)a$	$(r^2+6r+8)n+(2r+3)a$
	$-((r^3/3)+(9r^2/2)$	$-((r^3/3)+(9r^2/2)$
	$-(5r/6)-2)$	$-(5r/6)-1)$
QMR:	$35n+9a+16$	$24n+4a+18$
Jacobi:	$11n+4a-1$	$5n+2a$
SOR:	$10n+5a-1$	$4n+2a$

Tabla 3.2: Métodos iterativos. Modelo de FLOPs para los métodos iterativos, siendo n la dimensión de la matriz, a el número de entradas no nulas de la matriz y r el parámetros restart para el método GMRES.

precond	FLOPs (construcción)	FLOPs (aplicación)
Jacobi:	n	n
Neumann:	a	$2ga+gn$
SSOR:	$(3a-n)/2$	$2a-n$
ilu0:	$(a-n)/2+(a/n)(a-n)/2$	$2a-n$
iluT:	$2a+(2n^3-n^2+n)/2$	$2a+(2n_p-1)n$
	$+log(n_p+a)$	
lsquares:	$a+(11(g+1)+19)(g+1)/2$	$2g(a+n)$

Tabla 3.3: Precondicionadores. Modelo de FLOPs para los precondicionadores, siendo n la dimensión de la matriz, a el número de entradas no nulas de la matriz, g el grado del polinomio en los precondicionadores de Neumann y Mínimos Cuadrados, y n_p el número de entradas nuevas permitidas por fila en la descomposición ILU(t).

talizar las rutinas de pase de mensajes y ejecutar para distintas configuraciones del sistema paralelo.

Se analizan las comunicaciones presentes en el producto matriz dispersa-vector de la implementación MPI del núcleo, que está esquematizado en la figura 3.4 y representado visualmente en la figura 3.5. El comportamiento del producto matriz dispersa vector desde el punto de vista de las comunicaciones se divide en tres fases:

Primera. Redistribución del vector por bloques a cíclico por columnas. En primer lugar hay una etapa de preprocesamiento en la que se determina dónde se ubican los datos que se van a enviar a cada procesador, con que *stride* y qué numero de ellos. Este procesamiento sólo se ejecuta una vez al principio del método y, dada la regularidad de las distribuciones, carece

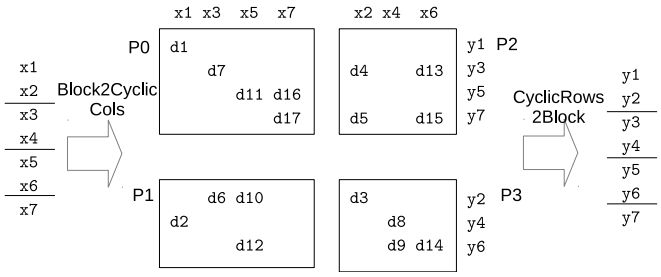

Figura 3.5: Producto matriz dispersa-vector en MPI en una red de 2×2 procesadores.

de comunicaciones. Para realizar la redistribución se efectúa una etapa de comunicación por filas y, a continuación, otra por columnas para completar los datos que necesita cada procesador. Cada una de estas etapas corresponde a una llamada a una operación de comunicación colectiva en MPI: `alltoallv` por filas y `allgatherv` por columnas (figura 3.6).

Segunda. Reducción para obtener el vector de salida en todos los procesadores (distribución cíclica por filas): `allreduce`.

Tercera. Redistribución del vector de salida de cíclico por filas a bloques. Vuelve a ser necesaria una etapa de preprocesamiento en la que se determina la estructura el flujo de información, otra vez carente de comunicaciones. En este caso, y puesto que hay redundancia de datos, hay elementos que pueden ser descartados y sólo es necesaria una etapa de comunicación por columnas. Corresponde a una comunicación colectiva de tipo `alltoallv` (figura 3.7).

3.4.1. Comunicaciones colectivas

Como acabamos de comentar, la reimplementación de los métodos de la librería se han utilizado tres operaciones de comunicación colectiva. Cada una de ellas se utiliza varias veces, tanto en las redistribuciones de datos entre MPI y HPF como en los núcleos. Las operaciones son:

- `alltoallv`, donde se intercambian datos dirigidos por un vector de índices
- `allgatherv`, donde se hace una recolección de datos para repartirlos posteriormente
- `allreduce`, una reducción con una operación de suma

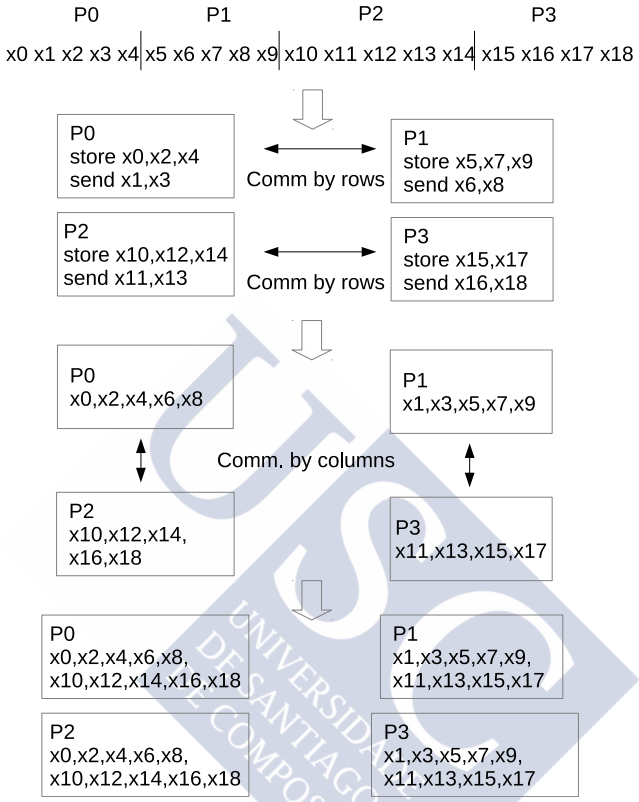

Figura 3.6: Redistribución de un vector (N=19) de bloques a cíclico por columnas en una red de 2×2 procesadores.

En la literatura hay varios modelos para la implementación eficiente de estas colectivas como árboles binarios, modelo postal [109], LogP [142] o LogGP [133], y algunas específicas de la topología de red, como mallas, toroides o hipercubos [128][164]. En cualquier caso, una implementación que minimice el coste de estas operaciones depende fuertemente de diversos parámetros de la arquitectura y de cómo se ajuste el modelo abstracto de máquina paralela a la usada para la implementación del algoritmo.

En cualquier caso, el coste de las operaciones colectivas que se precisa caracterizar puede describirse como una combinación de ciertos bloques básicos: `broadcast`, `reduce` y `gather/scatter` [168][26][27].

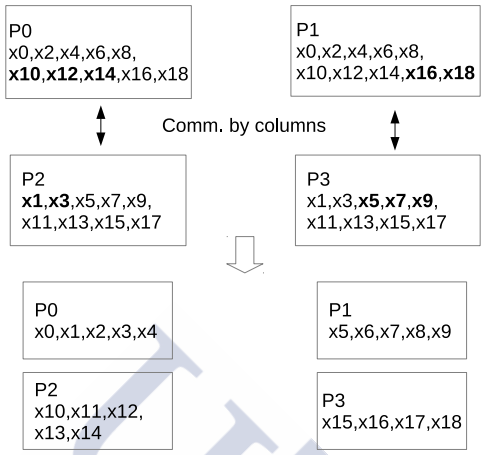

Figura 3.7: Redistribución de un vector (N=19) de cíclico por filas a bloques en una red de 2×2 procesadores.

Operación `broadcast`. Envía un dato de un procesador determinado al resto de procesadores que se usan en el ámbito de la operación. La implementación tradicional en un multiprocesador se basa en un modelo de árbol *minimum spanning tree*, donde, recursivamente, la difusión agrupa los nodos restantes en dos subconjuntos y envía los datos necesarios sólo a los representantes de cada subconjunto hasta que se cubran todos los nodos del sistema. Para p procesadores el coste de la operación se puede modelar por la siguiente expresión:

$$t(n) = \lceil log_2(p) \rceil \cdot (a + b \cdot n) \tag{3.1}$$

donde a y b caracterizan la comunicación punto a punto usando un modelo lineal, siendo n es el tamaño del vector.

Operación `reduce`. Obtiene un único resultado a partir de los datos distribuidos en todos los nodos, aplicando una operación de combinación (aritmética en este caso). Las operaciones son similares a las de difusión, pero en orden inverso, realizando en cada caso la operación aritmética correspondiente. El coste de la operación puede expresarse del siguiente modo:

$$t(n) = \lceil log_2(p) \rceil \cdot (a + b \cdot n + c \cdot n) \tag{3.2}$$

donde c representa el coste de la operación de combinación.

Operación `gather` o `scatter`. Se implementa de forma análoga al `broadcast`, pero en cada paso sólo son transferidos los datos con origen o destino en el subconjunto de nodos. Si

el reparto de datos entre nodos está equilibrado, el coste puede caracterizarse por la siguiente ecuación:

$$t(n) = \lceil log_2(p) \rceil \cdot a + ((p-1)/p) \cdot b \cdot n \tag{3.3}$$

Combinando estos bloques podemos obtener expresiones algebraicas que modelen las operaciones colectivas presentes en PARAISO. Los parámetros de ajuste se obtienen mediante un ajuste por mínimos cuadrados sobre los tiempos de ejecución en sistemas reales.

Operación alltoallv

Se utiliza esta función en las redistribuciones de datos, usando comunicadores para operar sólo entre las filas o columnas de la malla de procesadores. El modelo analítico es conceptualmente equivalente a la concatenación de un `gather` de datos seguido de un `broadcast`. Usando los bloques básicos, el modelo queda del siguiente modo:

$$t(n) = 2 \cdot \lceil log_2(p) \rceil \cdot a + ((p-1)/p + \lceil log_2(p) \rceil) \cdot b \cdot n \tag{3.4}$$

Para aplicar estas ecuaciones, se deben realizar ajustes por tramos para mensajes pequeños, intermedios y grandes para encontrar los coeficientes. Los límites dependen de la arquitectura concreta del sistema paralelo y están definidos por parámetros de la arquitectura enmascarados por el modelo. Las variaciones obtenidas entre el modelo y la realidad dan una medida de los efectos de la arquitectura no modelados por esta ecuación de coste: topología de red, jerarquía de memoria, efectos del sistema operativo, etc. Dependiendo del tamaño de mensaje n, los parámetros a, b y c tendrán más o menos peso en la función de coste.

Operación allgatherv

Se utiliza esta función en las redistribuciones de datos de bloques a cíclico por columnas y a cíclico por filas. El modelo corresponde a una operación de recolección donde el resultado queda almacenado en todos los nodos, dirigido por un vector de índices. Resulta similar a `alltoallv`, pero en este caso se envía el mismo conjunto de datos a todos los nodos:

$$t(n) = 2 \cdot \lceil log_2(p) \rceil \cdot a + ((p-1)/p + \lceil log_2(p) \rceil) \cdot b \cdot n \tag{3.5}$$

La diferencia entre ambos modelos aparece en la interpretación de los parámetros a y b, que reflejan la diferencia entre acceder siempre a los mismos datos, como sucede en `allgatherv`, o acceder a diferentes conjuntos de datos, en `alltoallv`.

Operación allreduce

La operación de reducción es una suma de los productos parciales que se han calculado localmente en cada procesador dentro del núcleo del producto disperso matriz-vector, y el resultado está accesible en todos los nodos. El modelado se hace en base a una reducción más una difusión y el coste es equivalente a:

$$t(n) = 2 \cdot \lceil log_2(p) \rceil \cdot a + ((p-1)/p + \lceil log_2(p) \rceil) \cdot b \cdot n + \lceil log_2(p) \rceil \cdot c \cdot n \qquad (3.6)$$

Para localizar el valor de los coeficientes se hace un ajuste en dos fases. Primero se ajusta la ecuación $2 \cdot \lceil log_2(p) \rceil \cdot a + K(p) \cdot n$ para varios valores del número de procesadores. En un segundo paso se ajuste $K(p)$ sobre el número de procesadores p.

Con las ecuaciones obtenidas en los apartados anteriores para computaciones y comunicaciones se puede obtener una estimación completa del coste. Estos valores se desviarán de los reales debidos a eventos no considerados, como costes de acceso a memoria, tiempos de espera en comunicaciones o fallos cache. En particular, el modelo para la predicción del rendimiento refleja una estimación optimista, pues no considera los tiempos de espera y desincronizaciones propios de la implementación MPI de los núcleos básicos.

Un análisis exhaustivo de la modelización de la librería PARAISO puede encontrase en la bibliografía [186].

USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA

Capítulo 4

Simulación de contaminación atmosférica en arquitecturas distribuidas

Este capítulo examina un programa de simulación de contaminación atmosférica basado en modelos meteorológicos, centrándose en la rutina que consume la mayor parte de su tiempo de ejecución. Se presenta brevemente el problema al que está orientada la simulación, para continuar con un análisis del código que ha permitido identificar la parte más costosa. Una vez determinada la rutina a estudiar, se ha llevado a cabo el modelado de sus computaciones y comunicaciones.

4.1. Contexto

Consideramos los modelos de calidad del aire que se utilizan para el control de las emisiones producidas por fuentes contaminantes como un caso particular de los problemas de ciencia e ingeniería que procesan gran cantidad de datos, realizan cálculos computacionalmente costosos y que requieren representaciones complejas para visualizar los resultados.

El objetivo de estas herramientas es encontrar una relación entre las sustancias que son emitidas a la atmósfera, debidas a la actividad humana o causas naturales, y las concentraciones de estas sustancias, y otras originadas por ellas, en la atmósfera. El análisis de esta relación es esencial en los estudios de contaminación atmosférica. La potencia de cálculo de los siste-

mas actuales hace posible construir modelos elaborados que consideran el comportamiento de los procesos físicos, químicos y biológicos reales involucrados, sin necesidad de considerar simplificaciones excesivas, y con un elevado nivel de precisión.

En este campo, una de las soluciones más extendidas para alcanzar este objetivo son los modelos constituidos por conjuntos de ecuaciones que describen los aspectos físicos y químicos del proceso y que, basándose en la representación matemática de los diferentes fenómenos de transporte y transformación que puede experimentar, tratan de calcular la distribución de una sustancia que se emite a la atmósfera. Matemáticamente, estas aplicaciones requieren la integración de un sistema de ecuaciones diferenciales ordinarias no lineales acopladas, que se resuelven, por ejemplo, utilizando un método de diferencias finitas. Estos métodos son computacionalmente costosos y la paralelización es fundamental para obtener tiempos de ejecución adecuados en aplicaciones reales.

STEM-II (*Sulphur Transport Eulerian Model 2*) es un modelo de calidad del aire que simula la dispersión de elementos contaminantes a través de la atmósfera [57][55][56]. En este capítulo se extrae un modelo computacional de la versión paralela de este código para caracterizar y evaluar su rendimiento.

De manera independiente, este modelo también puede ser utilizado en estrategias de balanceo dinámico de la carga para determinar buenas distribuciones del espacio de simulación entre los recursos disponibles. El problema del balanceo es crítico para la aplicación de este código. Para obtener el rendimiento óptimo es necesario realizar un balanceo efectivo de la carga del programa entre los diferentes nodos de computación, evitando que alguna de las unidades de procesamiento esté ociosa. Encontrar una solución óptima al balanceo es un problema NP completo, por lo que es habitual hacer uso de soluciones heurísticas y aproximadas, que se pueden obtener aplicando estos modelos.

4.2. STEM-II

Las centrales térmicas constituyen una de las fuentes de contaminación atmosférica más significativas en la actualidad. Para llevar a cabo el control y prevención de la contaminación generada y minimizar el impacto ambiental es necesario un control estricto del proceso de producción, pero también resulta imprescindible la capacidad de simular escenarios posibles

sobre la base de los datos reales. Para ello se necesitan modelos eficientes de los procesos involucrados.

Los modelos STEM fueron desarrollados para proporcionar una base teórica en el estudio de las relaciones existentes entre las emisiones de contaminantes, el transporte atmosférico y los procesos químicos y de transformación, y la distribución resultante de los contaminantes en el aire, así como, los patrones de deposición. STEM-II es un modelo numérico que no está limitado a una escala o región específica, excepto por los datos de entrada disponibles y por que los modelos físicos, químicos y meteorológicos son relativos a ciertas escalas de tiempo y de longitud.

Este modelo necesita como datos de entrada información meteorológica como datos sobre el viento, nubes, temperatura, precipitaciones, etc., junto con el inventario de emisiones, velocidades de deposición, topografía y condiciones límite iniciales y finales. La fuente que proporciona la información meteorológica es independiente del modelo. Como resultado, el modelo proporciona las concentraciones de las especies tanto en fase gas como en fase líquida, las velocidades de reacción, los flujos hacia el interior y el exterior del dominio, la cantidad de especie depositada y las concentraciones de radicales e iones.

El modelo matemático en que se basa STEM-II se describe por medio de ecuaciones diferenciales parciales tridimensionales y dependientes del tiempo. La ecuación base es la ecuación de difusión atmosférica. La solución de esta ecuación requiere la integración de un sistema de ecuaciones diferenciales ordinarias no lineales y acopladas. STEM-II resuelve este sistema utilizando un método de diferencias finitas, que implica la definición de una malla tridimensional, de modo que las variables del modelo se calculan en cada punto de esa malla.

El modelo se ha utilizado para verificar el impacto ambiental originado por una central térmica (en concreto, la localizada en As Pontes, A Coruña) en la producción de energía eléctrica por medio de la combustión de carbón [74]. Debido a la alta humedad presente en el entorno de la central y a las elevadas concentraciones de SO2 en las emisiones, era necesario modelar procesos químicos en fase gas y en fase líquida para llevar a cabo un estudio completo del comportamiento de los contaminantes. El entorno de interés para la central cubría una superficie de 61×61 km^2 centrada en las instalaciones, con resolución de 1×1 km^2. Verticalmente, alcanzaba los 4200 metros de altura, distribuidos en 15 niveles.

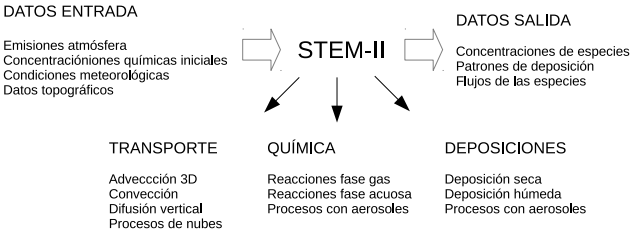

Figura 4.1: Esquema funcional del modelo de difusión atmosférica STEM-II.

STEM-II tiene una estructura modular que le permite realizar, de manera separada, los distintos pasos que comprende la simulación final de dispersión de contaminantes (figura 4.1). Los módulos son:

- Módulo de transporte. En este bloque funcional se tienen en cuenta la emisión de polutos desde fuentes puntuales y zonales, el transporte por advección, convección y difusión turbulenta, y la variación espacial y temporal del viento, temperatura, presión, vapor de agua, precipitaciones y los campos de nubes.
- Química. Este bloque engloba la modelización del desarrollo diurno y la disipación de la capa de mezcla aérea, el tratamiento detallado de la topología y la radiación solar, y la transformación de especies químicas en fases líquida y gaseosa.
- Deposiciones. Este bloque realiza la simulación del movimiento de polutos a niveles bajos y a través de nubes, así como los procesos de precipitación de los mismos (deposición seca y húmeda). STEM-II es un modelo diseñado para tratar procesos de deposición con detalle.

4.3. El código de la aplicación

STEM-II usa un gran número de variables: concentración de especies, temperatura, velocidad del viento, índice de precipitaciones, etc. El programa necesita matrices de gran tamaño para almacenar esta información en cada punto de la malla de simulación. La memoria total que necesita el programa es proporcional al tamaño de la malla de simulación, siendo, además, una aplicación computacionalmente muy intensa [205][171].

```
STEM-II:
    i1 <- calcular datos independientes del tiempo
    for t in tiempo_de_simulación:
        i2 <- calcular datos dependientes del tiempo
        transporte_horizontal()
        vertlq:
            for loop_y in 1..nlz:
                for loop_x in 1..ix:
                    if iclfl:
                        asmm()
                        vertcl()
                        for loop_trfl in 1..mend:
                            vertcl()
                            rxn()
                        clean
                    else:
                        vertcl()
                        rxn()
        entrada/salida
```

Figura 4.2: Pseudocódigo del programa STEM-II, donde *iclfl* corresponde a la comprobación *fase_gas()* y *fase_acuosa()*. Los módulos más relevantes en la evaluación del rendimiento son *asmm()*, que calcula parámetros de la interconversión de fases, *vertcl()*, que resuelve las ecuaciones del transporte vertical para especies gaseosas y *rxn()*, que calcula las reacciones químicas en el eje z.

El programa se divide, según su funcionalidad, en tres grandes bloques: el módulo de transporte horizontal, el módulo *vertlq* y el módulo de entrada y salida (figura 4.2). En el módulo de transporte horizontal se calcula el transporte de especies en las dimensiones horizontales *x, y*. La química más el transporte vertical se simulan en el módulo *vertlq*. Las principales rutinas de este módulo son tres: *asmm*, que calcula los parámetros asociados a las nubes, *vertcl*, que realiza los cálculos correspondientes al transporte vertical, y *rxn*, que implementa las operaciones relacionadas con la química del programa. Por último, el módulo de entrada y salida se encarga de proporcionar los datos necesarios para realizar los cálculos y devolver los resultados.

STEM-II presenta una estructura de lazos similar a cualquier simulación basada en el método de diferencias finitas. El programa secuencial consiste principalmente en cinco lazos anidados, que destacan por su importancia en el peso computacional de la aplicación. El más externo de estos lazos, *loop_t*, especifica la duración temporal del proceso de simulación, ya que cada iteración del lazo simula un minuto de tiempo real. Dentro del módulo *vertlq* se encuentran

los lazos *loop_x* y *loop_y* que recorren las dimensiones horizontales del espacio simulado, así como un lazo interno, *loop_trfl*, que controla la velocidad de las reacciones en fase líquida. Un último lazo, *loop_z*, se encuentra en el interior de la subrutina *rxn* y recorre la dimensión espacial vertical.

La versión original del programa no es paralelizable de manera inmediata, ya que el lazo temporal debe ser ejecutado de forma secuencial. Este lazo presenta dependencias verdaderas entre iteraciones consecutivas, de manera que los resultados obtenidos en una iteración son la base para los cálculos de la siguiente.

El módulo *vertlq* es la parte computacionalmente más costosa del programa y la paralelización se ha centrado en ella [170]. Este módulo se ejecuta una vez en cada iteración del lazo temporal y se encarga de realizar los cálculos relacionados con el transporte de las especies en la dimensión espacial vertical y con la química de los procesos considerados en el modelo. Recorre las dimensiones horizontales (*x, y*) a través de los dos lazos externos *loop_x* y *loop_y*. En cada iteración de estos dos lazos se realizan operaciones relacionadas con la dimensión vertical asociada a las coordenadas, es decir, se trabaja con la columna de puntos de la malla 3D de simulación. Así mismo, se determina el tipo de fase que se debe emplear, gaseosa o líquida. Las operaciones relacionadas con la fase líquida tienen una mayor complejidad y peso computacional que las relacionadas con la fase gaseosa. Los lazos paralelizables de este módulo, *loop_x* y *loop_y*, están perfectamente anidados y no presentan dependencias entre iteraciones. Para poder aprovechar mejor de la jerarquía de memoria, se ha encontrado que es preferible paralelizar únicamente el lazo más externo de los dos, *loop_y*.

Dentro de *vertlq* (figura 4.3), la rutina *rxn*, que se encarga de la química del problema, es, con diferencia, la que consume la mayor parte de los operaciones en punto flotante del módulo, utilizando típicamente del orden del 80 % del tiempo total de ejecución del programa secuencial [160]. Pero este valor es muy dependiente de los datos meteorológicos de entrada. En particular, el número de puntos de la malla de simulación en los que haya que emplear la fase líquida determinará realmente el peso computacional de *rxn* en el programa, ya que los cálculos relacionados con la química en esta fase son mucho más costosos que los correspondientes a la fase gas. Por tanto, la importancia de esta subrutina en el comportamiento del programa viene determinado, en gran medida, por el tamaño del lazo que establece el número de veces que se ejecuta *rxn* dentro de *vertlq*, que está directamente relacionado con las reacciones químicas que tienen lugar en esta fase.

```
vertlq:
    for loop_y in 1..nlz:
        for loop_x in 1..ix:
            if iclfl:
                asmm()
                liqic()
                vertcl()
                cover()
                timset()
                for loop_trfl in 1..mend:
                    for iph in 1..3:
                        trcoed()
                        vcoel1()
                        decomp()
                    for l in 1..iliq:
                        vcoel2()
                        trid1()
                    rxn()
                clean
            else:
                liqic1()
                vertcl()
                cover()
                rxn()
```

Figura 4.3: Esquema de la estructura interna de la rutina *vertlq*.

La estructura interna de *rxn* es muy irregular y compleja, y presenta un gran número de saltos condicionales (figura 4.4). Además, el número de iteraciones del principal lazo interno de esta subrutina depende fuertemente de los parámetros de entrada. Para el modelado de su comportamiento, por su importancia, se tiene que considerar el lazo externo *loop_z* que recorre la dimensión vertical, once bifurcaciones y un bucle.

Realizando un análisis de las dependencias de datos de STEM-II se obtiene que la rutina *rxn* puede paralelizarse en los tres lazos espaciales (x, y, z). Sin embargo, en un multicomputador de memoria distribuida la paralelización de *loop_z* no es rentable debido a la sobrecarga introducida por las comunicaciones que necesita [171]. En el esquema del código paralelo se ve que, antes de comenzar con la región paralela del programa, las matrices que contienen datos dependientes del tiempo son distribuidos entre los diferentes procesos. Por su parte, los datos independientes del tiempo son distribuidos antes de comenzar el lazo temporal ya que se mantienen constantes durante toda la ejecución. Cada procesador ejecuta el mismo código

```
rxn:
    for loop_z in 1,iz:
        racoe1()
        if CRS:
            aqupha()
            if CE: sion()
            if RE: sion()
            aqupha()
        reactn()
        state()
        if CR:
            for loop_tr in 1..mend1:
                if C: liqrxn()
                if R: liqrxn()
                if LC:
                    trans1()
                else:
                    if LR:
                        trans1()
                    else:
                        trans2()
        if CR:
            if C: sion()
            if R: sion()
```

Figura 4.4: Esquema de la estructura interna de *rxn*.

paralelo sobre sus datos, y finalmente las matrices modificadas en la parte paralela son recogidas para continuar con la parte secuencial del código. Este esquema se muestra en la figura 4.5.

Una de las conclusiones más importantes de este estudio es el hecho de que todas las variables que influyen en el comportamiento de estos bucles y bifurcaciones están relacionados exclusivamente con parámetros meteorológicos, circunstancia que denota la relevancia de las condiciones meteorológicas en el flujo de ejecución. El significado de los parámetros que controlan las bifurcaciones están relacionados con la concentración de agua en las diferentes fases, debido a que, como ya se comentado, el coste computacional de la rutina *rxn* es mucho más elevado para la fase líquida que para la fase gaseosa.

```
STEM-II se ejecuta en el proceso raíz
scatterv: datos independientes del tiempo
for t in tiempo_de_simulación:
    scatterv: datos dependientes del tiempo
    ejecutar en secuencial: transporte_horizontal()
    scatterv: datos modificados
    ejecutar en paralelo en cada procesador: vertlq()
    gather: recuperar datos modificados
```

Figura 4.5: Esquema de comunicaciones entre procesadores para una ejecución paralela del código STEM-II. Hay una sección de código que se calcula secuencialmente en el procesador raíz, para realizar a continuación un reparto de la malla de simulación entre procesadores y ejecutar vertlq independientemente en cada uno.

4.4. Caracterización de las computaciones

En este análisis se utilizó la librería PAPI [7][47] para acceder a los contadores hardware y modelar el comportamiento de las regiones críticas del código. PAPI es altamente portable está y bien soportada bajo Fortran, lo que resulta necesario para el código de STEM-II

Es factible abordar la modelización del tiempo de ejecución de *vertlq* a partir de los FLOPs de *rxn*, puesto que existe una relación lineal entre ambos. Por lo tanto, para conseguir una ejecución paralela balanceada es necesario obtener, para cada coordenada (x, y), una aproximación de los FLOPs ejecutados por la rutina *rxn*. Este resultado será empleado más adelante como referencia en el reparto de tareas para conseguir una distribución balanceada de la carga computacional.

Debido a la complejidad de su estructura interna (figura 4.6) no es fácil predecir el coste computacional de la rutina *rxn*. El flujo principal de ejecución de la rutina es el principal factor a tener en cuenta para estimar su volumen computacional. Se identificaron 9 parámetros relacionados con los bucles y bifurcaciones más importantes de la rutina, aparte del lazo *loop_z*, con los que se puede predecir el comportamiento del flujo de ejecución principal. Estos parámetros tienen un carácter binario en el caso de las bifurcaciones, evaluando como 1 si se cumple la condición para bifurcar o con 0, en caso contrario. En el caso de los bucles, los parámetros, reflejan el número de iteraciones. Estos parámetros presentan una fuerte relación entre ellos ya que las condiciones de bifurcación son muy similares entre ellas o, en ocasiones, antagónicas.

```
rxn:
    for loop_z in 1,iz:
        racoe1()
        if [P0]:
            aqupha()
            if [P1]: sion()
            if [P2]: sion()
            aqupha()
        reactn()
        state()
        if [P3]:
            for loop_tr in 1..[P8]:
                if [P4]: liqrxn()
                if [P5]: liqrxn()
                if [P6]:
                    trans1()
                else:
                    if [P7]:
                        trans1()
                    else:
                        trans2()
        if [P3]:
            if [P4]: sion()
            if [P5]: sion()
```

Figura 4.6: Esquema de la estructura interna de *rxn*, en la que se muestran los parámetros P_i que controlan el flujo principal de ejecución.

A priori no se puede identificar la influencia de estos parámetros en el comportamiento de la rutina. Pero conociendo estos parámetros, es posible caracterizar el flujo principal de ejecución de *rxn*.

Por lo tanto, es necesario, por un lado, desarrollar un procedimiento que permita determinar, con un bajo coste asociado, los valores de estos parámetros antes de ejecutar la parte paralela del código y, por otro lado, obtener la relación entre los parámetros binarios y los FLOPs ejecutados por *rxn*. Realizando un análisis de las variables que determinan el comportamiento de bifurcaciones y bucles, se puede comprobar que éstas dependen únicamente de los datos de entrada meteorológicos y de los datos independientes del tiempo. Por tanto es factible precalcular los parámetros antes de realizar el reparto de la carga, siempre que se puedan conocer los datos meteorológicos correspondientes al instante simulado emulando la ejecución de *vertlq*, pero utilizando tan sólo las operaciones relacionadas con el cálculo de las

P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	FLOPs	desv
0	0	0	0	0	0	0	0	0	7715	724
1	0	0	0	0	0	0	0	0	10337	15
1	0	1	1	0	1	1	0	4	25531	1455
1	0	1	1	0	1	1	0	12	46298	2503
1	0	1	1	0	1	1	0	20	59153	979
1	0	1	1	0	1	1	0	60	167840	11262
1	1	0	1	1	0	0	1	4	21825	148
1	1	0	1	1	0	0	1	12	42885	707
1	1	0	1	1	0	0	1	20	60489	203
1	1	0	1	1	0	0	1	60	159078	2733
1	1	1	1	1	1	0	0	4	32124	972
1	1	1	1	1	1	0	0	12	66474	1743
1	1	1	1	1	1	0	0	20	97780	1722
1	1	1	1	1	1	0	0	60	264797	5139

Tabla 4.1: Media de los FLOPs de cada iteración del lazo externo *loop_z* de *rxn* para las distintas combinaciones de parámetros P_i.

variables que afectan a los parámetros binarios. El volumen computacional de la operación es despreciable frente al total de la rutina *vertlq*.

Para obtener la relación entre los parámetros y los FLOPs de *rxn*, se ejecutaron simulaciones utilizando datos meteorológicos en los que están representados una amplia variedad de situaciones climatológicas. Para cada iteración de *loop_z* en *rxn* se obtuvo el número de FLOPs y los valores de los parámetros binarios.

Un primer análisis de los resultados refleja que, efectivamente, existe una fuerte relación entre los parámetros, ya que sólo aparecen 14 combinaciones diferentes que se repiten a lo largo de la ejecución del programa. Para distintas iteraciones del lazo con la misma combinación de parámetros, se observa que los FLOPs medidos presentan valores muy similares, con una desviación típica siempre menor del 10 %, por lo que podemos considerar la media aritmética de todos los valores asociados a cada una de las combinaciones como representante de cada combinación. De esta forma, asociamos directamente cada combinación de parámetros con su valor medio de FLOPs. Esta información se muestra en la tabla 4.1.

Aún se puede hacer una simplificación más en los parámetros, ya que se aprecian claramente patrones. Un análisis de la tabla muestra que con tan sólo cuatro parámetros es posible deter-

P_0	P_1	P_2	P_8	FLOPs	desv
0	0	0	0	7715	724
1	0	0	0	10337	15
1	0	1	4	25531	1455
1	0	1	12	46298	2503
1	0	1	20	59153	979
1	0	1	60	167840	11262
1	1	0	4	21825	148
1	1	0	12	42885	707
1	1	0	20	60489	203
1	1	0	60	159078	2733
1	1	1	4	32124	972
1	1	1	12	66474	1743
1	1	1	20	97780	1722
1	1	1	60	264797	5139

Tabla 4.2: Media de los FLOPs de cada iteración del lazo externo *loop_z* de *rxn* para las distintas combinaciones de parámetros P_i simples.

```
if P0==0:
    FLOPs_z = 7715
else:
    if P1==0 and P2==0:
        FLOPs_z = 10337
    else:
        if P1==0 and P2==1:
            FLOPs_z = 2500*P8 + 13000
        if P1==1 and P2==0:
            FLOPs_z = 2440*P8 + 12400
        if P1==1 and P2==1:
            FLOPs_z = 4150*P8 + 15700
```

Figura 4.7: Modelo del número de operaciones en punto flotante en un punto de la malla de simulación.

minar el comportamiento de *rxn* en FLOPs. Estos parámetros son P_0, P_1, P_2 y P_8, y producen la simplificación que aparece en la tabla 4.2.

Cuando P_1 o P_2 toman un valor distinto de 0, el valor de los FLOPs asociados aumenta proporcionalmente a P_3. Así, a partir de los resultados obtenidos experimentalmente es posible estimar el valor de FLOPs para cada iteración del lazo *loop_z* de *rxn*, una vez conocidos los valores de estos cuatro parámetros, aplicando el algoritmo de la figura 4.7.

El número total de operaciones para una ejecución de *rxn* será la suma de los valores calculados para cada una de las iteraciones del lazo *loop_z*. Por otra parte, teniendo en cuenta que cada ejecución de *rxn* está asociada a una coordenada (x,y) del espacio simulado, los FLOPs de *vertlq* estimados para cada coordenada se obtienen multiplicando los FLOPs calculados para una ejecución de *rxn* por el número de veces que se ejecuta esta subrutina. Los parámetros P_i dependen únicamente de los datos meteorológicos y éstos permanecen constantes durante una ejecución de *vertlq*. El valor de la variable que determina el número de ejecuciones de *rxn* dentro de *vertlq* depende exclusivamente de la meteorología y de datos atemporales, y puede obtenerse de manera simultánea a la determinación de los valores de dichos cuatro parámetros.

Al haber utilizado contadores hardware, los valores numéricos de las expresiones de la figura 4.7 tienen una dependencia directa con el tipo de arquitectura utilizada para realizar las medidas de los FLOPs. Si fuera necesario utilizar el modelo en otras arquitecturas habría que recalcular los coeficientes de las expresiones, ya que sí se mantiene la dependencia entre el tiempo de ejecución y los FLOPs. Un análisis detallado de la obtención de estas ecuaciones puede encontrarse en la bibliografía [161][108][171].

4.4.1. Balanceo de carga

La primera versión paralela de STEM-II adopta una estrategia estática para el reparto de tareas entre los procesadores, centrándose en el paralelismo de los lazos externos de la rutina *vertlq* que recorren las dimensiones espaciales horizontales *x, y*. El principal problema que presenta la estrategia de reparto equitativa es que el comportamiento de *vertlq* depende fuertemente de los datos de entrada meteorológicos, que varían temporal y espacialmente a lo largo de la ejecución del programa. Por tanto, aunque todos los procesadores actúen sobre el mismo volumen de datos, el tiempo y los FLOPs consumidos pueden resultar muy dispares, dando lugar a fuertes desbalanceos. Es difícil conseguir un balanceo de carga computacional repartiendo equitativamente las coordenadas *x*, y del espacio simulado sin una caracterización más profunda del comportamiento de las computaciones [110][177][103][89][63][152].

Para lograr un balanceo adecuado de la sección paralela de STEM-II es necesario un mecanismo que permita, antes de realizar el reparto de tareas, estimar el tiempo de ejecución de cada una de las unidades indivisibles de datos que se reparten entre los procesadores, es decir, el tiempo asociado a cada coordenada *(x, y)*. De esta manera se puede determinar cuál

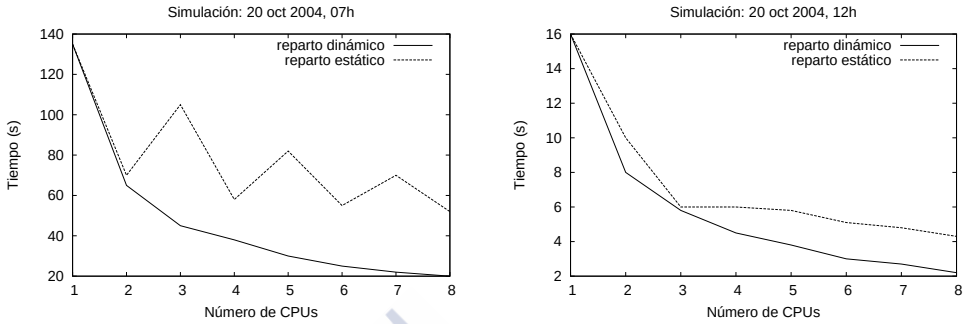

Figura 4.8: Comparación entre los tiempos de ejecución obtenidos mediante reparto estático y reparto dinámico para diferente número de procesadores.

es la mejor distribución de estos bloques para conseguir un reparto compensado de la carga computacional. El uso de este mecanismo no puede degradar el rendimiento del programa de manera que la carga computacional que suponga debe ser mínima.

Usando el modelo que acabamos de describir para realizar estimaciones del número de FLOPs para cada coordenada *(x, y)* puede abordarse el problema del balanceo de carga de manera dinámica para la rutina *vertlq*, analizando los datos meteorológicos antes de ejecutar el reparto de tareas. Dado que la estimación de FLOPs en cada punto de la malla de simulación varía únicamente en función de los datos meteorológicos que no cambian de manera brusca a lo largo del tiempo, los desbalanceos de carga se producirán de manera lenta y progresiva.

La heurística desarrollada para balancear la carga está basada en una distribución de la carga computacional de las columnas de la malla de simulación, dimensión *y*, para posteriormente hacer otra distribución equilibradas de la carga de las filas de cada una de las particiones resultantes, dimensión *x*. En la figura 4.8 se muestran las diferencias, en tiempo de ejecución, entre la estrategia de reparto estático inicial y la nueva estrategia de balanceo dinámico de la carga.

4.5. Caracterización de las comunicaciones

Como ya se ha mencionado, la primera versión paralela de STEM-II adopta una estrategia estática para el reparto de tareas entre los procesadores, centrándose en el paralelismo de los lazos externos de la rutina *vertlq* que recorren las dimensiones espaciales horizontales *x, y*.

En la práctica, y como ya se ha comentado, para obtener un mayor aprovechamiento de la localidad espacial de los datos, se ha optado por paralelizar únicamente la dimensión espacial *y*, por lo que todos los vectores que tienen una dependencia esta dimensión son distribuidos entre los elementos de procesamiento. Con este reparto estático el número de comunicaciones es fijo y depende del tamaño del problema y de las características de la arquitectura paralela.

Para cada instante temporal de la simulación, el código de STEM-II necesita ejecutar secciones de código secuencial, relacionada con el transporte horizontal, por contener dependencias que no pueden ser resueltas de manera eficiente en paralelo. Este código es ejecutado por el nodo raíz. Al terminar, desde este nodo raíz, se actualizan datos en los procesadores a través de una comunicación colectiva. Inicialmente, el nodo raíz ya ha enviado a los procesadores datos que son estables durante toda la ejecución. Los procesadores ejecutan simultáneamente, en paralelo, el código de *vertlq* y, al terminar, informan al procesador raíz de los nuevos valores en los puntos de simulación con otra operación colectiva. El proceso continúa hasta que se alcanza el límite temporal preestablecido.

Esto significa que, en esta versión paralela de STEM-II, las comunicaciones son debidas a:

- el envío de información inicial desde el procesador raíz al resto
- en cada instante temporal, el envío de nueva información calculada desde el procesador raíz al resto
- en cada instante temporal, la recepción de nueva información en el procesador raíz originada en los otros procesadores

La cantidad de información que se intercambia depende del número de puntos de simulación asignados a cada procesador. Esta distribución es estática, por lo que el patrón de comunicaciones se mantiene estable a lo largo del proceso. Cada procesador recibe aproximadamente el mismo número de puntos de simulación. Para un sistema de computación de N procesadores, la malla de simulación se reparte equitativamente en la dimensión y, de manera que a cada uno le corresponden aproximadamente $(dim\_y/N) \cdot dim\_x \cdot dim\_z$ puntos de simulación.

Considerando $M1$, el volumen inicial de comunicaciones originadas en el procesador raíz, $M2$, el volumen de comunicaciones entre el procesador raíz y el resto en cada instante temporal, y $M3$, el volumen de comunicaciones entre los procesadores y el procesador raíz en cada

instante temporal, y que al menos las dos últimas son proporcionales al volumen de simulación asignado a cada procesador, las comunicaciones entre el procesador raíz y otro cualquiera siguen el esquema siguiente:

$$\begin{aligned} \text{comunicaciones}(\text{procesador } i) \sim M1 + t \cdot M2 \cdot M3 \sim \\ M1 + t \cdot (dim\_y/N) \cdot M \end{aligned} \tag{4.1}$$

donde hemos agrupado en una constante M otros términos.

El volumen de comunicaciones entre el procesador que ejecuta el código secuencial y cualquier otro procesador sigue un comportamiento lineal en el tiempo durante la ejecución de una simulación. Debido al reparto estático elegido, con una distribución equitativa de puntos de simulación por procesador, el volumen de comunicaciones entre ellos es inversamente proporcional al número de procesadores en el sistema; considerando el sistema en conjunto, el volumen de comunicaciones es el mismo.

Para la nueva estrategia de paralelización, donde en cada procesador se asigna un número diferente de puntos de simulación, las comunicaciones entre el procesador con el código secuencial y cualquier otro sigue una relación más compleja. Sin embargo, considerando el sistema en conjunto, el volumen de comunicaciones es el mismo de antes. Para el sistema completo, el número de comunicaciones dependen principalmente del número de puntos de la simulación, no de cómo estén repartidos entre los procesadores.

En resumen, en este capítulo hemos presentado una modelización de *vertlq*, el bloque computacional más costoso de STEM-II, y se ha obtenido un conjunto de ecuaciones dependientes únicamente de los datos de entrada meteorológicos, para predecir la distribución de carga computacional de la rutina *vertlq* en las dimensiones espaciales horizontales *x, y* para un instante de simulación determinado. Debido a su pequeño coste computacional este mecanismo puede ser reaprovechado como elemento de decisión en una estrategia de balanceo dinámico para conseguir un reparto equilibrado de la carga computacional entre los elementos de computación disponibles. El volumen de comunicaciones del modelo, tanto en el reparto estático como en el dinámico, depende únicamente de la estructura de la malla de simulación.

Capítulo 5

Optimización estocástica en arquitecturas distribuidas

Este capítulo tiene como objetivo paralelizar y modelizar el rendimiento de un algoritmo estocástico de optimización usado en una herramienta de ordenación del territorio para la formulación de planes de uso del suelo. Para comenzar, se muestran brevemente los fundamentos teóricos y características particulares de esta implementación, para continuar con la identificación, análisis y modelado de las secciones útiles con las que caracterizaremos computaciones y comunicaciones. En este caso, el algoritmo de partida es secuencial, por lo que se propone y modela una paralelización en MPI a sistemas multiprocesador.

5.1. Contexto

El problema de la gestión del territorio, en un sentido amplio, involucra tanto factores sociales, económicos y políticos, como técnicos. El objetivo último es una ocupación útil del espacio con una mejor localización de asentamientos humanos y actividades para un desarrollo sostenible. La tarea requiere un enfoque interdisciplinar.

El diseño de planes de usos del suelo es la herramienta básica para adaptar físicamente el entorno y situar los usos en las mejores localizaciones posibles atendiendo principalmente a su aptitud, a la vez que se maximizan objetivos relacionados con aspectos económicos, sociales o ecológicos particulares de la zona, y se respetan todas las restricciones presentes en el área destino.

La complejidad del proceso de planificación aumenta por la necesidad de realizar la tarea considerando un número cada vez mayor de objetivos en conflicto. También por la tendencia actual en favor de facilitar la participación pública, lo que exige una mayor publicidad y justificación de las decisiones. Por este motivo, desde hace años se han desarrollado herramientas que asisten a los gestores en el proceso de diseño de planes y que implementan nuevas técnicas basadas en metodologías de evaluación multicriterio. La disponibilidad creciente de información geoespacial de calidad ayuda a que la información necesaria para el análisis y modelado del problema sea accesible y precisa.

Tal vez la tecnología más representativa de este campo sean los sistemas de información geográfica (SIG). Estos sistemas representan la integración de manera organizada de hardware, software y modelos geográficos. Han sido diseñados para capturar, almacenar, consultar, manipular, analizar y desplegar, en todas sus formas, información con una componente espacial asociada con el fin de resolver problemas complejos de gestión. Desarrollos comunes en ciencias de la computación hacen que estas herramientas sean cada vez más amigables, más potentes, y puedan almacenar y procesar más información.

Pero el proceso de planificación de usos tiene otras necesidades. Requiere además información del pasado, presente y futuros posibles, y metodologías con las que incorporar metas, objetivos, costes y beneficios a los modelos [46][111][124]. Utilizando los sistemas de información geográfica como plataforma de desarrollo, los nuevos sistemas de apoyo a la planificación combinan datos espaciales, no espaciales y temporales junto con modelos y simulaciones para identificar las relaciones causa-efecto entre todos los factores involucrados, cuantificar sus impactos, evaluar las implicaciones de la selección de uno u otro curso de acción, formular soluciones alternativas y mostrar resultados de simulación del territorio en el futuro [146][147]. A veces incluyen también soporte de colaboración entre individuos y grupos, para facilitar el diálogo y la negociación entre las partes.

Una de estas herramientas es RULES, desarrollada por el Laboratorio del Territorio de la USC para asistir en el proceso de la planificación del territorio [191][199][192]. Nuestro interés por ella deriva del hecho de que uno de los algoritmos que están disponibles para resolver el problema de la localización espacial de los usos es el *Simulated Annealing* (SA), que condiciona el rendimiento de la herramienta y para el que buscamos un modelo en este capítulo.

RULES integra tres etapas de un proceso de planificación de usos del suelo rural, de manera que los resultados de una etapa puedan ser usados como entradas para una etapa posterior.

Para conseguir eso, varios algoritmos están disponibles en cada uno de los tres módulos que implementan las etapas del proceso (ver figura 5.1):

1. Módulo de evaluación de tierras. Incluye tres técnicas para evaluar la aptitud del terreno. Dos de ellas son métodos de análisis multicriterio: suma lineal ponderada [83][165] y análisis del punto ideal [54]. El tercer método es el esquema FAO con limitaciones en el sistema de puntuaciones [219][91]. El resultado de cualquiera de estas tres técnicas es un mapa continuo de aptitud con valores reales entre 0 y 1.

2. Módulo de optimización del área. Este módulo resuelve un modelo de programación lineal donde las variables de decisión corresponden a los usos del suelo y las funciones objetivo incluyen la maximización del margen bruto, empleo rural, área cultivada y grado de naturalidad de la vegetación, así como la minimización de los costes de producción y el uso de productos agroquímicos. El usuario gestor asigna prioridades a cada uno de esos objetivos. El resultado de cualquiera de las cinco técnicas disponibles en el módulo, con restricciones a priori, a posteriori o interactivas, es una lista numérica del volumen de área propuesta para cada uso del suelo considerado [64][117].

3. Módulo de localización espacial. Su objetivo es diseñar el mapa final de usos a partir de los mapas de aptitud y áreas recomendadas calculados en los módulos anteriores. Incluye tres métodos de ubicación espacial: optimización jerárquica, el análisis del punto ideal [54] para objetivos en conflicto y un algoritmo heurístico basado en el Simulated Annealing (SA) que es el objetivo específico de este trabajo [145].

5.2. Algoritmo

Debido al lenguaje común matemático un amplio número de modelos de distintas áreas de la ciencia y la ingeniería llevan a la aparición de problemas de optimización donde se necesita identificar la mejor solución, es decir, el valor óptimo. Dada una función f que representa el problema, la formulación básica es:

> Minimizar la función f sujeta a las restricciones $\{h_i\} \leq 0$, donde f y $\{h_i\}$ son funciones definidas con variables continuas o discretas en el dominio $\mathbb{R}^m \rightarrow \mathbb{R}$.

Los problemas que modelan sistemas reales habitualmente están caracterizados por la no convexidad del dominio de búsqueda o de la función objetivo. Además se asume que las funciones

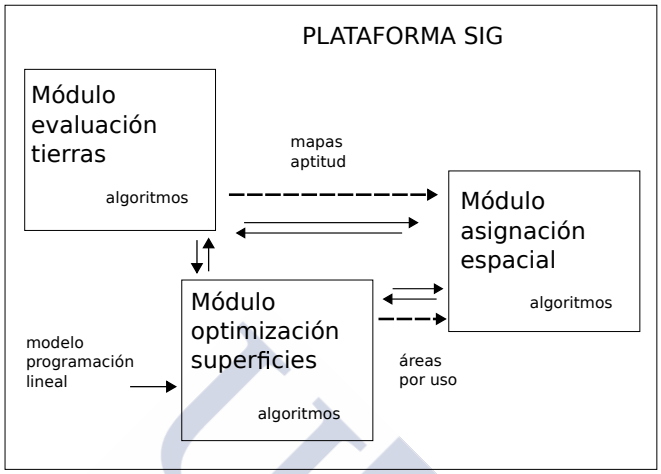

Figura 5.1: Esquema de RULES.

f y h_i pueden ser evaluadas en todos los puntos; es decir, que para cada posible combinación de las variables de entrada se puede acceder al valor de la función mediante sus expresiones analíticas, si éstas están presentes, o por medios empíricos [181]. Por último, el problema se simplifica en aquellos casos donde las pendientes de estas funciones están acotados por un límite superior finito (constante de Lipschitz) [226].

Para estas condiciones, frecuentes en la práctica, utilizamos una descripción alternativa del problema de búsqueda del óptimo:

> Dado un espacio de configuraciones finito (espacio de búsqueda) $S = \{x|x = (x_1, x_2 \ldots x_m)\}$, $S \in \mathbb{R}^m$, y una función de coste que asigna un número real a cada configuración, $f' : S \to \mathbb{R}$, se quiere encontrar una configuración óptima $x*$, $x* \in S$, tal que $\forall z \in S$, $f'(x*) \leq f'(z)$.

La definición se refiere a la optimización como un proceso de minimización del valor de la función. Puede también plantearse trivialmente como un problema de maximización con cambios mínimos. Vamos a hablar de forma general de optimización sin especificar la dirección.

Las aproximaciones deterministas a la optimización explotan propiedades analíticas del problema como la convexidad o monotonicidad para generar una secuencia determinística de

puntos (finita o no) que convergen al valor óptimo [181]. La alternativa son los métodos estocásticos de optimización, que se basan en perturbaciones aleatorias, y que se utilizan para problemas que no tengan una estructura evidente que pueda ser explotada analíticamente. Estos métodos requieren poca información extra del problema y su convergencia es probabilística [226].

En los algoritmos de optimización estocástica basados en el gradiente de coste, una primera fase del algoritmo genera un conjunto de puntos a partir de una distribución uniforme sobre la superficie de búsqueda o la función de coste [125]. En la siguiente fase, un procedimiento de búsqueda local por descenso del gradiente se aplica a cada uno de esos puntos, produciendo varios óptimos locales. El mejor óptimo local es el óptimo global encontrado. El inconveniente principal de estas técnicas es que las búsquedas locales pueden quedar atrapadas en mínimos locales. Esto es, quedar bloqueadas en un entorno de soluciones en el que todo movimiento lleva a soluciones peores, sin poder alcanzar la solución óptima que se encuentra fuera de ese entorno. Para minimizar este efecto, los algoritmos se ejecutan desde diferentes puntos de inicio elegidos aleatoriamente.

El algoritmo de búsqueda comienza en un estado inicial aleatorio [233]. Genera un estado vecino, o bien aleatoriamente o seleccionando el vecino más prometedor tras una búsqueda exhaustiva, y lo evalúa. Si hay una reducción en el coste se realiza la transición entre estados (el estado actual se reemplaza con el nuevo estado). En caso contrario se mantiene el estado original. Y este proceso se repite hasta que no puedan ser encontradas mejoras entre los vecinos del estado actual. Así, el algoritmo termina en un mínimo:

```
1 estado <- configuración inicial aleatoria
2 hacer
3     estado' <- transformar(estado, vecino_aleatorio())
4     si coste(estado') <= coste(estado)
5         estado <- estado'
6 hasta que coste(estado') > coste(estado) para todo estado vecino
```

Este algoritmo termina necesariamente en el primer mínimo local que encuentra, lo que puede dar lugar a un coste ciertamente alejado del coste mínimo global. La razón por la que se queda bloqueado es que sólo acepta transiciones que disminuyen el coste de la solución actual.

El algoritmo SA, una metaheurística estocástica de optimización, es una generalización de esta búsqueda local que introduce un elemento de aleatoriedad en el proceso de búsqueda y aceptación para evitar los mínimos locales. Funciona con un mecanismo de evaluación de transiciones que acepta, además de movimientos que corresponden a una mejora de la función de coste, también algunos movimientos que empeoren el valor de dicha función, si bien estos últimos son aceptados con una probabilidad finita y controlada que disminuye en el tiempo. La combinación de ambos tipos de movimiento permiten al algoritmo desplazarse por estados que no mejoran directamente la solución, y que permiten escapar de los mínimos locales y recorrer más regiones del espacio de búsqueda [145][166][59].

La base del mecanismo de evaluación de transiciones en el algoritmo está basado en un procedimiento introducido por Metropolis para simular los estados de equilibrio de un sistema físico a una temperatura dada [166]: en cada paso, se genera al azar una perturbación pequeña de la configuración del sistema y se evalúa el cambio resultante en la energía del sistema, ΔE. La nueva configuración es aceptada con probabilidad 1 si $\Delta E \leq 0$ y con probabilidad $\exp(-\Delta E/kT)$ si $\Delta E > 0$. La distribución de probabilidad en el equilibrio coincide con la distribución de Boltzmann.

Cuando se aplica a problemas de optimización, el concepto de energía se sustituye por una función objetivo de coste, y se utiliza la temperatura T como parámetro de control del ratio de aceptaciones. K juega el papel de una constante de normalización.

Una descripción general [233] del algoritmo de Metropolis es la siguiente:

```
estado <- configuración inicial aleatoria
t <- temperatura
hacer
    estado' <- transformar(estado, movimiento_aleatorio())
    d <- coste(estado') - coste(estado)
    si random(0, 1) <= min(1, exp(-d/(k*t)))
        estado <- estado'
hasta alcanzar situación de equilibrio
```

El algoritmo SA incorpora el concepto de temperatura al procedimiento básico de búsqueda local. El sistema comienza con una temperatura elevada, T, y se aplica un esquema de enfriamiento para disminuirla de acuerdo a un procedimiento preestablecido. En cada etapa de temperatura T se generan series de nuevos estados aleatorios a partir de perturbaciones del

estado actual y se simula el sistema por el procedimiento de Metropolis hasta que se alcanza el equilibrio. Según la temperatura disminuye, los estados que empeoran la función de coste tienen mayor probabilidad de ser rechazados y el proceso eventualmente termina en un estado con el coste óptimo o próximo al óptimo.

Una descripción general del algoritmo es la siguiente:

```
estado <- configuración inicial aleatoria
t <- temperatura inicial
hacer
    hacer
        estado' <- transformar(estado, movimiento_aleatorio())
        d <- coste(estado') - coste(estado)
        si d <= 0
            estado <- estado'
        en caso contrario
            si random(0, 1) <= min(1, exp(-d/(k*t)))
                estado <- estado'
    hasta alcanzar situación de equilibrio
    t <- actualizar(t)
hasta alcanzar criterio de parada
```

Desde el punto de vista teórico, dado un estado x, se genera un nuevo estado $x*$ que es aceptado con una probabilidad que coincide con el criterio de Metropolis.

$$\text{mín}(1, \exp((f(x*) - f(x))/T)) \tag{5.1}$$

El SA y sus variantes [85][185] han sido extensamente aplicados en distintos dominios para problemas de optimización. Una ventaja importante de este método es que no requiere conocimiento especializado sobre cómo resolver el problema, lo que le permite ser aplicado a diferentes ámbitos sin cambiar la estructura computacional básica [208]. Se puede demostrar que el proceso converge a una solución arbitrariamente próxima al mínimo global.

Dado su interés interdisciplinar, el SA ha sido abordado de diferentes maneras:

1. Siguiendo la analogía entre encontrar un estado de mínima energía en un sistema físico y encontrar la configuración de menor coste en un problema de optimización combinacional. Éste es la aproximación original del trabajo de Metropolis y ha originado el vocabulario especializado del algoritmo tomado de la mecánica estadística [166].

2. Considerando el SA como uno de los métodos heurísticos para manejar problemas de decisión complejos (junto con otras heurísticas como algoritmos genéticos, redes neuronales, búsqueda con tabú y análisis de objetivos, entre otros). Este es precisamente el tipo de problema que interesa en la planificación del territorio [220][190][15][202].

3. Considerando el SA como un algoritmo de aleatorización en términos de un autómata estocástico con o sin capacidades de aprendizaje.

Con este último planteamiento, la prueba formal de convergencia del algoritmo al óptimo global puede ser establecida utilizando la teoría de cadenas de Markov. La secuencia de perturbaciones (o movimientos) que son aceptados por el algoritmo forman una cadena de Markov donde el siguiente estado depende sólo del estado actual, no estando influenciado por la historia pasada. Así, la probabilidad de siguiente estado $i+1$ es el producto de dos probabilidades: la probabilidad de que el estado $i+1$ sea generado por un movimiento desde el estado i y la probabilidad de aceptar el estado $i+1$ [201][33][113].

El diseño eficiente del algoritmo requiere:

- una función de coste bien formulada
- un mecanismo para generar un número de reordenamientos aleatorios (perturbaciones, movimientos o iteraciones) con los que explorar el espacio de búsqueda
- la selección de un calendario de enfriamiento para controlar la temperatura durante el proceso
- la condición de terminación del algoritmo

La función de coste

Es un diseño dependiente de la aplicación que mide el valor relativo de un estado (una solución) respecto a otro. Asigna un valor a cada configuración del sistema y tiene que ser construida de modo que sistemas claramente desventajosos se evalúen como más costosos que sistemas claramente más ventajosos. Además, cuanto más adecuado sea el sistema como solución, menor coste asociado debe tener. En algunas aplicaciones esta función puede ser de naturaleza analítica, lo que significa que su forma puede ser determinada a priori. En otros casos es necesario determinarla de manera indirecta a partir del proceso en ejecución y medirla empíricamente.

La definición e implementación de la función de coste es uno de los factores que más influye en el tiempo de ejecución del algoritmo puesto que se debe evaluar sobre cada transición propuesta por el sistema de generación de movimientos [93].

El mecanismo de generación de movimientos

Determina el concepto de transición entre estados en la aplicación, y de vecindad entre estados (estados accesibles en un movimiento). Es otro factor importante en el tiempo de ejecución del algoritmo, siendo también muy dependiente del problema [134][137][218][114].

El calendario de enfriamiento

Es la secuencia de valores que toma el parámetro de temperatura, comenzando con valores altos para terminar con valor nulo [137][88][12]. Este parámetro está relacionado con el criterio de aceptación de Metropolis para transiciones que incrementen la energía del sistema: con valores altos de temperatura, la probabilidad de aceptación es alta, mientras que, para temperaturas próximas a cero, la aceptación es prácticamente nula. Nótese que la constante K también puede ser utilizada para modular este efecto.

Un elemento importante del calendario de enfriamiento es la elección del valor de temperatura inicial. Se selecciona un alto valor inicial para asegurar que la mayoría de los movimientos propuestos son aceptados. Esto permite una búsqueda más amplia en el espacio de soluciones al principio del proceso ignorando las estructuras presentes en el estado inicial. Sin embargo, un valor excesivo puede consumir demasiado tiempo del proceso antes de que el algoritmo comience a converger hacia el óptimo, por lo que se han propuesto reglas heurísticas para establecer un valor inicial; por ejemplo, en función de la tasa inicial de aceptaciones en las primeras iteraciones.

Para construir una secuencia de temperaturas que disminuya en el tiempo un procedimiento habitual es, a partir de la temperatura inicial, multiplicar en cada etapa la temperatura previa por un factor inferior a 1. En ese caso, el decrecimiento de la temperatura tiene un comportamiento exponencial, lo que ofrece la ventaja añadida de que se hace más lento según avanza el proceso de búsqueda y permite una mejor exploración del sistema en las etapas finales. Y un mayor número de temperaturas garantiza encontrar mejores soluciones [135].

El criterio de parada

El análisis del modelo matemático proporciona información sobre las condiciones necesarias y suficientes para garantizar la convergencia del algoritmo a una solución óptima global con probabilidad 1. Desafortunadamente, estas condiciones no pueden ser satisfechas en tiempo finito, y se necesita utilizar el calendario de enfriamiento para establecer valores para los parámetros, de manera que se puedan encontrar soluciones próximas a la óptima en tiempo finito, con un número finito de transiciones a cada valor de temperatura y una secuencia finita de valores de temperatura [135][222].

Se han propuesto distintos esquemas para determinar la temperatura de finalización del algoritmo [233]. Existe una temperatura a partir de la cual ya no se realizan mejoras en el proceso, dada por

$$T_f = (f'_m - f_m)/log(u) \tag{5.2}$$

donde f_m es el mínimo absoluto de la función de coste, f'_m es el siguiente mejor valor de la función de coste, y u es el número de movimientos necesarios para desplazarse desde f'_m a f_m. Este límite representa el escenario del peor caso de manera que, utilizando esta temperatura como criterio de terminación, el algoritmo no se detiene hasta encontrar f_m.

Alternativas a este criterio son definir un valor umbral conveniente para el coste del estado final, detener el proceso cuando no se hayan encontrado nuevas soluciones en un cierto número de etapas de temperatura, o utilizar un número prefijado de etapas de temperatura (es decir, utilizar un criterio por tiempo de ejecución en lugar de por número de soluciones). En cualquier caso, todas estas alternativas van a terminar el proceso de SA de forma prematura por lo que no garantizan llegar a la solución óptima [157].

La solución más inmediata para permanecer en la misma temperatura durante largos períodos de tiempo es utilizar un número de iteraciones fijo determinado por las características del problema y el tamaño del espacio de búsqueda. Cuántas más operaciones se realicen a una temperatura, y más despacio disminuyan las temperaturas, a priori mejor será la calidad de la solución obtenida. Para una temperatura dada, un número suficiente de iteraciones siempre alcanza el equilibrio, que es el punto en el que la distribución temporal de estados aceptados es estacionaria (y que corresponde a una distribución de Boltzmann). Sin embargo, esto no es siempre óptimo desde un punto de vista práctico y se ha experimentado con calendarios adap-

tativos donde el número de movimientos depende de la cantidad de aceptaciones y rechazos obtenidos hasta el momento [134][222].

5.3. Implementación

El algoritmo de SA considerado en este trabajo forma parte de la tercera etapa de la herramienta RULES, donde el usuario ya ha tomado decisiones sobre qué usos del suelo incorporar en la solución y en qué cantidades, y ha construido los mapas de aptitud necesarios. En este punto el algoritmo permite calcular dónde situar cada uso maximizando la aptitud del resultado final al mismo tiempo que cumple restricciones geométricas.

Particularizamos nuestro estudio a esta implementación concreta. No obstante el modelado se ha realizado de manera genérica para que el mismo proceso pueda aplicarse a otros algoritmos, aunque para ello se tengan que adaptar aspectos específicos del modelo, que dependen del problema que se está resolviendo, como es la codificación del estado actual, la construcción de la función de coste y el mecanismo de generación de movimientos.

El estado actual

El primer aspecto a considerar, la representación interna de los estados, es una interpretación directa de la información espacial como formato raster. Porque a pesar de trabajar sobre una plataforma SIG y, por tanto, poder procesar directamente geometrías en formato vectorial, las técnicas empleadas en RULES están enfocadas a formatos raster. En éstos los datos vectoriales son proyectados sobre una cuadrícula que representa las unidades mínimas del territorio como celdas individuales, identificadas por una pareja de coordenadas con valores enteros. El contenido de cada una de estas celdas representa el valor promedio de una variable espacial determinada entre los límites marcados por el tamaño de la celda; y, en caso de que sea necesario, ponderados por el área del territorio incluido en ella.

En el caso de los mapas de aptitud de usos construidos, la variable representada es la calidad del terreno para cada uno de los usos del suelo considerado. Cada celda contendrá un valor real entre 0 y 1, siendo 1 el valor óptimo, que se obtiene en la primera etapa de la herramienta.

Por sencillez, la malla de rasterizado es una superficie rectangular que se superpone al territorio en estudio usando celdas cuadradas (ver figura 5.2). Estas celdas presentan ventajas importantes frente a otras alternativas. Por una parte, identificar una celda concreta se hace

simplemente por dos valores enteros (las coordenadas x, y). Por otra, es más sencillo calcular las vecindades de una celda, ya sea en la versión de cuatro vecinos: vecino al norte, al este, al sur y al oeste; o en la versión de ocho vecinos: vecino al norte, al noreste, al este...

En principio no todas las celdas de la malla tienen que estar disponibles para el algoritmo, sino que puede haber algunas que no se puedan usar por estar fuera del área de estudio (cuando el área de estudio es más pequeña que la malla) o porque sus posiciones están prohibidas por algún motivo técnico (fronteras, zonas de protección o presencia de infraestructuras, entre otros). También los límites de una celda pueden contener territorio dentro del área de estudio y territorio fuera del área de estudio. En ese caso el valor almacenado en la celda es ponderado proporcionalmente al área de estudio incluida en la celda.

El tamaño de la celda de rasterización define la precisión alcanzada en la solución final, y es un compromiso entre la precisión frente a la capacidad de cálculo.

Por último, la superposición de todos los mapas debe dar una intersección no nula, esto es, existir un conjunto de celdas que contengan valores en cada mapa de aptitud. Si alguna de las celdas de un mapa de entrada no contiene información, la celda con esas coordenadas se elimina de todos los mapas de entrada. Así, el territorio en estudio se identifica por la intersección de todos ellos.

El estado actual es representado internamente por un raster donde el valor de cada celda corresponde al identificador del uso que el algoritmo le ha asignado. También se reserva un identificador para indicar que la celda está fuera del área de estudio (figura 5.3).

Por cuestiones de eficiencia, otras variables que derivan del estado o de las condiciones iniciales se almacenan y mantienen continuamente actualizadas. Esto sucede para la evaluación de coste del estado actual, los parámetros internos a cada término de la función de coste, o los coeficientes de normalización.

El estado final es el estado al que se llega cuando el algoritmo alcanza la condición del criterio de parada. Es, por tanto, una evolución del estado inicial y se almacena con el mismo formato.

El hecho de que la solución pueda expresarse como un raster donde cada celda contiene un valor (el identificador de uso) y está conectada con otras celdas (vecindad de la celda), permite expresar el problema de localizar las celdas más adecuadas a cada uso como un problema de

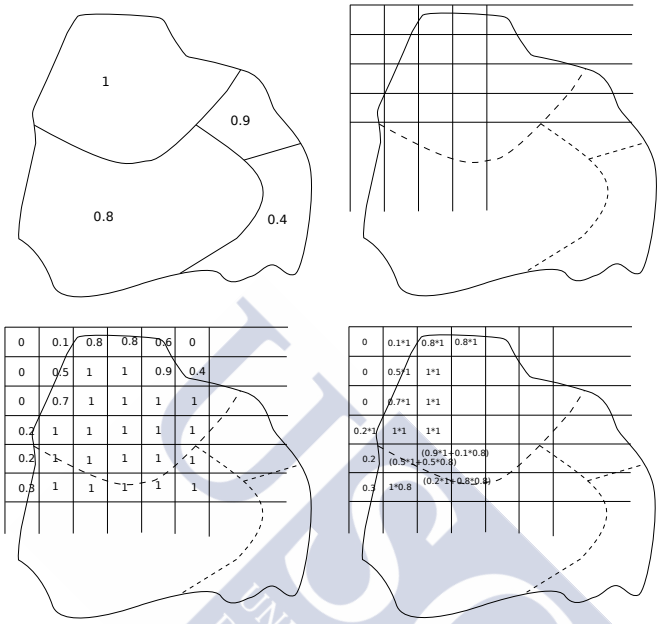

Figura 5.2: Proceso de rasterización. El mapa vectorial original se superpone con la malla de rasterización, se genera una máscara de ocupación y el resultado se pondera por los valores de aptitud en cada región.

```
estado actual:

    int[][] mapa de usos
    double  coste del estado

    término t1 de la función de coste:
        double  aptitud actual
        double  aptitud máxima global
        double  coeficiente normalización t1

    término t2 de la función de coste:
        int     perímetro actual
        int[]   perímetro actual por cada uso
        int     perímetro mínimo global
        double  coeficiente normalización t2
```

Figura 5.3: Representación del estado actual.

particionamiento de grafos, donde no se conoce el número de particiones, pero sí el tamaño total de las particiones de cada uso (el número de celdas solicitadas por uso).

La función de coste

El diseño de la función de coste de un algoritmo de SA permite introducir la evaluación de múltiples subobjetivos especializados simultáneamente en el proceso de optimización. La implementación propuesta de la función de coste global está definida como una combinación lineal de dos términos independientes, de manera que cada uno de ellos evalúa un subobjetivo del problema geoespacial. De esta manera se pueden utilizar los coeficientes de la combinación como parámetros de control del proceso de evaluación, lo que permite al usuario establecer la influencia de cada subobjetivo en el coste total. El primero de los términos está destinado a maximizar la aptitud del sistema y el segundo a maximizar la compacidad de cada uno de los usos:

$$c = \sum_i (r_i \cdot termino_i) \tag{5.3}$$

siendo c el valor de la función de coste, r_i el coeficiente de ponderación del término i y $termino_i$ su valor.

La aptitud de una configuración se calcula como la suma de valores de aptitud de cada celda. Se trata de un número decimal entre 0 y 1 que se obtiene del mapa de aptitud del uso asignado a la celda.

Hay varias alternativas para calcular la compacidad de la configuración. En aplicaciones geográficas es una práctica habitual utilizar como medida de compacidad el área de una parcela dividido por su perímetro al cuadrado. Minimizando este parámetro, denominado circularidad, se priman las formas circulares para las parcelas ya que esta geometría obtiene el valor mínimo de $\pi/4$.

Sin embargo, ya que el área asignada a cada uso no varía a lo largo del proceso, en RULES se optimiza la compacidad de la solución mediante la minimización de los perímetros. De esta manera se evitan cálculos costosos en punto flotante. Otra simplificación importante es que se ignora la existencia de grupos de parcelas del mismo uso y se tratan todos los perímetros del uso como si pertenecieran a una única parcela grande. Esto vuelve a mejorar el rendimiento evitando los cálculos para comprobar si se han creado nuevas parcelas del uso o eliminado alguna después de cada transición.

Antes de combinar ambos términos, los valores se normalizan y se aprovecha para establecer los signos adecuados en las ecuaciones (el término de aptitud aumenta cuando la aptitud del sistema, como suma de aptitudes de celda, se acerca al óptimo; sin embargo la compacidad del sistema, como suma de compacidades de cada uso, disminuye):

$$\begin{aligned} \text{aptitud\_normalizada} &= \frac{(\text{aptitud\_máxima} - \text{aptitud})}{(\text{aptitud\_máxima} - \text{aptitud\_mínima})} \\ \text{compacidad\_normalizada} &= \frac{(\text{compacidad} - \text{compacidad\_mínima})}{(\text{compacidad\_máxima} - \text{compacidad\_mínima})} \end{aligned} \tag{5.4}$$

No se necesitan valores exactos de aptitud y compacidad máximas y mínimas, sino que bastan cotas de referencia que no puedan ser superadas. Se elige como aptitud máxima el valor que resulta de sumar la aptitud del mejor uso en cada celda. Este valor es probablemente inalcanzable porque no tiene en consideración que cada uso está limitado a utilizar N_i celdas.

Se usa el mismo método para calcular una aptitud mínima: la suma del menor valor de aptitud una vez que se revisan todos los usos en cada celda.

Para la compacidad máxima se consideran las compacidades parciales de cada uso en su configuración óptima: un cuadrado. Así, para un uso de N_i celdas, se asume que se distribuyen en un cuadrado de lado $\sqrt{N_i}$ formando un perímetro de $4\sqrt{N_i}$. Esta configuración es probablemente inalcanzable porque estos cuadrados óptimos difícilmente cubrirán geométricamente toda el área de estudio.

Por otro lado, la compacidad mínima se obtiene del caso en que cada celda forma su propia parcela, con tantas parcelas finales como celdas. Para un uso con N_i celdas, la compacidad mínima viene dada por el perímetro de N_i celdas independientes, $4 \cdot N_i$. Tampoco es probable que esta configuración se alcance en un caso real.

La generación de movimientos

El proceso de generación de transiciones en cada iteración del SA realiza una modificación aleatoria mínima sobre el estado que da lugar a un nuevo estado del espacio de búsqueda, localizado en la vecindad del anterior. El mecanismo para generar transiciones es el siguiente: desde el estado actual se seleccionan al azar dos celdas que cumplan que los valores de uso asignado sean diferentes y que ninguna celda tenga valor nulo para el uso. A continuación se intercambian los valores de uso de las celdas entre sí.

Este tipo de transiciones implica que no se modifica el número de celdas asignadas a cada uso. El número de celdas por uso se mantiene constante durante todo el proceso e igual al número de celdas objetivo por uso, que es un dato de entrada.

Realizar una transición intercambiando el uso de dos celdas aleatorias implica cambios en la función de coste. Una transición modifica los dos subobjetivos, aptitud y compacidad, pero sólo en los dos usos involucrados. Esta propiedad es un punto esencial para la evaluación de la función de coste global del sistema en base a medidas locales y permite una implementación eficiente de la forma en que se calculan los costes totales (mediante evaluaciones locales de resultados parciales) y una reducción importante en el tiempo de ejecución.

Al realizar la transición, si se tienen N usos, y para cada uno de ellos se lleva cuenta de los resultados parciales de aptitud así como también de su compacidad, una transición deja intactos los valores de aptitud y compacidad de $N-2$ usos, y sólo modifica los de los 2 usos intercambiados. Mantener actualizados estos resultados parciales de los términos evaluados en la función de coste cuando se modifica el estado, simplifica la reevaluación. Además, son actualizaciones computacionalmente poco costosas.

El calendario de enfriamiento

La implementación actual de RULES utiliza un calendario de enfriamiento exponencial. Esto permite especificar toda la secuencia de temperaturas con únicamente dos valores iniciales: una temperatura inicial y una constante de enfriamiento.

Cada temperatura nueva se establece a partir de la anterior multiplicando por una constante de enfriamiento fija ct con valor inferior a uno. La secuencia de valores del parámetro temperatura, $\{T_0 \cdot ct^i\}$, forma de esta manera una función exponencial decreciente que tiende asintóticamente a una temperatura cero, según muestra la figura 5.4. Debido a la falta de precisión numérica, esta temperatura cero puede ser alcanzada en un número finito de pasos.

El criterio de parada

La solución implementada en RULES para establecer el criterio de parada es indicar un número máximo de temperaturas que considerar y detener el proceso cuando se alcancen. Este número es proporcional al número de temperaturas necesarias para que el coste del estado

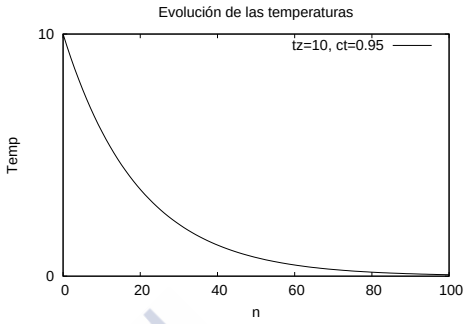

Figura 5.4: Evolución de la temperatura en el calendario de enfriamiento exponencial.

disminuya a la mitad en una ejecución modelo, y suficientemente elevado para que el proceso alcance una solución adecuada en el rango de valores utilizado para los parámetros de control.

De la misma manera, dentro de cada etapa del calendario de enfriamiento, tiene que poder realizarse un número suficientemente elevado de movimientos para llegar al equilibrio térmico, y así estar seguros (probabilísticamente) de no estar limitados por los mínimos locales. RULES ha elegido para este valor un número proporcional al número de celdas disponibles.

La elección del estado inicial

RULES ofrece la posibilidad de introducir un estado inicial del sistema como punto de partida del proceso de optimización, pero también permite generar automáticamente un estado inicial aleatorio. En ambos casos el estado inicial debe cumplir con la restricción de que cada clase de usos del suelo tenga asignadas tantas celdas como celdas objetivo se proponen. Como el proceso de generación de transiciones sólo intercambia el valor de usos entre celdas, esta condición tiene que cumplirse desde el principio.

Interesa modelar el caso con la solución inicial aleatoria por dos razones. Primero, porque es el uso básico de la herramienta RULES, que pretende obtener soluciones alternativas al plan de usos del suelo en lugar de corregir el plan actual. Y, segundo, porque evita tener que considerar el estado inicial como un dato de entrada para pasar a considerar simplemente el número de celdas de cada uso.

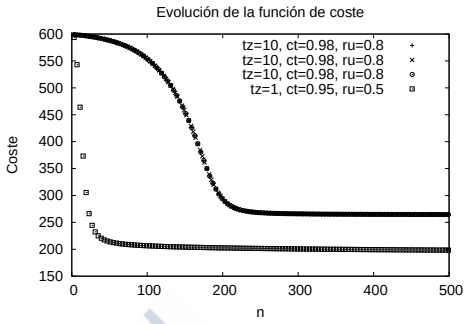

Figura 5.5: Evolución temporal de cuatro procesos de optimización. Los tres primeros procesos comienzan bajo las mismas combinaciones de parámetros de entrada (*tz*, *ct* y *ru*) y se superponen, mientras que el último comienza con una combinación de los parámetros de entrada distinta y sigue una evolución temporal diferente.

Aunque la heurística del SA es una optimización estocástica, después de realizar múltiples ejecuciones se encuentra que, bajo las mismas condiciones iniciales, los estados aleatorios evolucionan prácticamente igual hacia el mínimo. Es decir, que dentro del rango de valores considerados para los parámetros de control, reutilizando los parámetros de entrada y de control del algoritmo, la evolución temporal del algoritmo es muy similar, de manera que sí se pueden hacer afirmaciones generales sobre el proceso sin tener que especificar un estado inicial concreto. Es un resultado empírico relacionado con detalles de esta implementación: elevado número de movimientos por temperatura, elevado número de temperaturas y parámetros de control que favorecen altas tasas de aceptaciones en las primeras etapas.

A pesar de lo dicho las evoluciones temporales no son idénticas. Debido a la naturaleza estocástica del SA, el proceso de optimización no está determinado a priori y diferentes ejecuciones desde las mismas condiciones iniciales, con los mismos parámetros de control, proporcionan soluciones ligeramente diferentes, aunque cerca del óptimo. La figura 5.5 y la tabla 5.1 muestran un ejemplo de este comportamiento.

5.3.1. Ejemplo de ejecución

A continuación se muestra un ejemplo de la ejecución de la implementación del algoritmo descrito.

	valor *tz*	valor *ct*	valor *ru*	coste inicial	coste final
opción a	10	0.98	0.8	599.613	264.513
opción b	10	0.98	0.8	599.603	264.468
opción c	10	0.98	0.8	601.576	264.537
opción d	1	0.95	0.5	618.247	198.396

Tabla 5.1: Evolución temporal de cuatro procesos de optimización. Los tres primeros procesos comienzan bajo las mismas combinaciones de parámetros de entrada (*tz*, *ct* y *ru*) y alcanzan valores similares, mientras que el último comienza con una combinación de los parámetros de entrada distinta y sigue una evolución temporal diferente.

El usuario puede influir en la evolución del algoritmo a través de dos conjuntos de parámetros: ponderaciones para los subobjetivos de la función de coste y la definición del calendario de enfriamiento.

Los parámetros de entrada utilizados en este ejemplo son:

- 2695 filas
- 2846 columnas
- 4315411 celdas no nulas (56 % del total)
- 13 usos y mapas de aptitud:
  - eucalipto: solicitadas 193675 celdas
  - forrvp: 695875 celdas
  - frondosas: 690175 celdas
  - frutales: 5600 celdas
  - hortalizas: 363250 celdas
  - maiz: 757818 celdas
  - otrcereales: 3550 celdas
  - otrforr: 63125 celdas
  - pastizales: 115725 celdas
  - patata: 47700 celdas
  - prados: 774668 celdas
  - resinosas: 554025 celdas
  - trigo: 50225 celdas
- 500000000 posibles movimientos por cada temperatura
- 500 temperaturas

Por otro lado, se han elegido para la función de coste:

- término de aptitud: 0.8
- término de compacidad: 0.2

Por último, los valores seleccionados para el calendario de enfriamiento son:

- temperatura inicial: 10
- constante de enfriamiento: 0.98

En la figura 5.6 se muestra el resultado final como una distribución espacial de usos, mientras que la figura 5.7 muestra las evoluciones temporales de la función de coste, cada uno de los términos de la función, de la temperatura y del número de aceptaciones (tanto aceptaciones directas como aceptaciones por el criterio de Metropolis). Otros ejemplos, junto con un análisis detallado de los parámetros, se pueden encontrar en [115][192][190].

El proceso de optimización es una sucesión de etapas con el núcleo básico de operaciones: generar un nuevo estado a partir del anterior, evaluar su coste, aplicar el criterio de aceptación directa y de aceptación probabilística para aceptar o rechazar del nuevo estado. Durante la ejecución del algoritmo el estado actual se desplaza por la superficie de búsqueda del problema en busca de la mejor solución. Las dificultades de esta exploración quedan reflejadas en las transiciones evaluadas y se puede utilizar un *random walk* sobre el espacio de búsquedas a distintas temperaturas para visualizarla (figura 5.7).

5.4. Caracterización de las computaciones

Como se ha comentado previamente, el proceso de generación de transiciones en el SA realiza una modificación aleatoria mínima desde el estado actual que nos sitúa en un nuevo estado del espacio de búsqueda, evalúa la función de coste en el nuevo estado y la compara con el valor del estado anterior. En algunos casos acepta la transición y actualiza tanto el estado actual como el valor de energía del estado. En otras ocasiones rechaza la transición y regresa al estado previo. Este proceso se realiza un número prefijado de veces. A continuación se cambia el valor del parámetro de temperatura por el siguiente del calendario de enfriamiento, y se repite de nuevo el proceso. El algoritmo termina cuando se cumple el criterio de parada programado, que es haber recorrido un número predefinido de temperaturas.

Es decir, en cada transición propuesta la decisión de aceptar o rechazar el nuevo estado se realiza en dos pasos: (1) se aceptan todas aquellas que disminuyan la función de coste y (2) se acepta además cualquier otra con una probabilidad proporcional a $exp(-\Delta E/T)$.

Figura 5.6: Resultado de la ejecución modelo. Los usos han sido distribuidos por el territorio optimizando la aptitud y la compacidad de las parcelas.

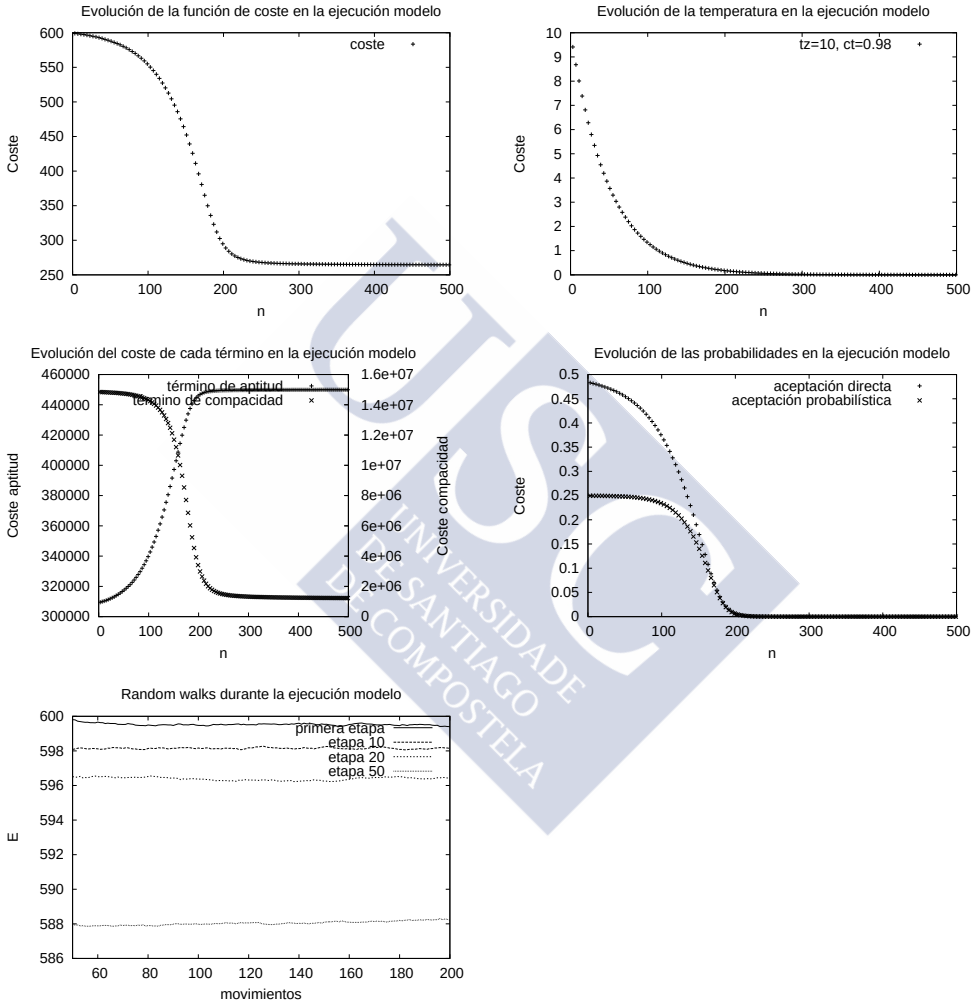

Figura 5.7: Evolución temporal de los parámetros internos en la ejecución modelo: función de coste, temperatura, la evolución de cada uno de los términos de la función de coste y las probabilidades de aceptación. Se muestran también diferentes *random walks* durante el proceso de optimización, correspondientes a los primeros movimientos a las temperaturas 10.0, 8.17, 6.67 y 3.64.

```
estado <- estado_inicial()
energía <- evaluar_coste(estado)
temperatura <- temperatura_inicial
hacer
    hacer
        estado' <- estado
        energía' <- energía
        estado'' <- transformar(estado', movimiento_aleatorio())
        energía'' <- reevaluar_coste(estado'')
        si energía'' < energía'
            aceptar_movimiento()
        en caso contrario
            d <- exp((energía'-energía'')/k*temperatura)
            si d<random(0, 1)
                aceptar_movimiento()
            en caso contrario
                rechazar_movimiento()
    durante M movimientos
    temperatura <- enfriar(temperatura)
durante N temperaturas

aceptar_movimiento() <-
    estado <- estado''
    energía <- energía''

rechazar_movimiento() <-
    estado <- estado'
    energía <- energía'
```

Figura 5.8: Esquema de la implementación del algoritmo de Simulated Annealing.

El pseudocódigo de la implementación del algoritmo es como aparece en la figura 5.8.

Puede apreciarse que las diferencias a nivel computacional entre aceptación directa de una transición, aceptación probabilística y el rechazo son escasas. En ambos casos el código sustituye el estado actual por alguna de las dos alternativas disponibles: el estado original o el nuevo estado. Y sustituir el estado implica cambiar tanto los usos de las celdas, como también otras variables precalculadas que se mantienen actualizadas para una evaluación eficiente de la función de coste. Hay pocas diferencias en el número de instrucciones enteras o flotantes entre una transición que se acepta y una que se rechaza. Las diferencias tienen que ver más con los test de aceptación que con el propio proceso de actualización.

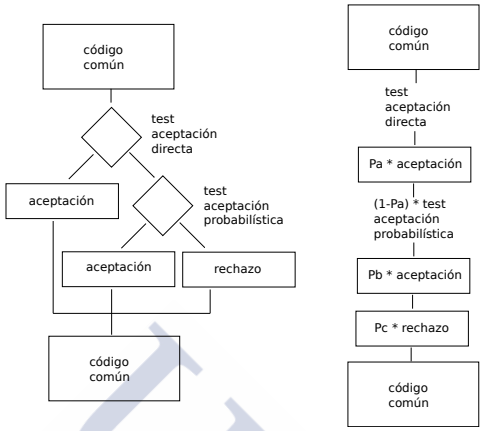

Figura 5.9: Esquema de las tres vías de ejecución del código en cada iteración, y el modelo global (con las probabilidades p_a, p_b y p_c).

Identificamos tres posibles vías de ejecución para el tratamiento de una transición (movimiento, iteración o perturbación),y que denominamos aceptación directa, aceptación probabilística y rechazo (figura 5.9).

Nótese que:

- La mayor parte del algoritmo es código común a todas las transiciones
- Toda transición ejecuta el test de aceptación directa independientemente del resultado
- Cada transición rechazada por el test de aceptación directa, ejecuta el test de aceptación probabilística independientemente del resultado

La principal dificultad para modelar este algoritmo estocástico se debe a que el código ejecutado depende fuertemente del valor concreto de los datos que se están procesando, los cuales se pueden conocer únicamente durante la ejecución. En cada instante debemos trabajar con suposiciones fundadas en los distintos valores estáticos que conocemos: tamaño de los mapas, número de celdas con valor y celdas nulas, número de usos, número de celdas solicitadas para cada uso, valores de los mapas de aptitud, calendario de enfriamiento, número de movimientos por temperatura o número de temperaturas.

Esta información es suficiente para caracterizar las instrucciones presentes en los fragmentos que hemos clasificado como código común, pero no para los test de aceptación y el código asociado a las tres vías de ejecución, que dependen de la evolución temporal de estado, que a su vez depende del proceso de convergencia del algoritmo de SA.

Es posible resolver este problema estableciendo, a partir del análisis de la evolución del algoritmo, unas probabilidades de aceptación directa y de aceptación probabilística, dependientes de las variables de entrada y de control del algoritmo y del instante temporal, que permitan reducir las tres vías de ejecución a un modelo que englobe todas las posibilidades.

Así, en lugar de tres vías alternativas tenemos una sola que, con el uso de las probabilidades p_a, p_b y p_c, integra todos los comportamientos (5.9):

$$p_a \cdot \text{aceptación directa} + p_b \cdot \text{aceptación probabilística} + p_c \cdot \text{rechazo} \tag{5.5}$$

Las probabilidades utilizadas son las siguientes:

- p_a, probabilidad de aceptación directa
- p_b, probabilidad de aceptación probabilística
- p_c, probabilidad de rechazo probabilística

Estas probabilidades cubren todo el rango de posibilidades, por lo que cumplen:

$$p_a + (p_b + p_c) = 1 \tag{5.6}$$

Para obtener los modelos del SA, en lugar de utilizar los contadores hardware, hemos explorado el código contabilizando los distintos tipos de operación. Cada línea del código contribuye a las instrucciones dependiendo (1) del tipo de operaciones en la línea, (2) del nivel de anidamiento en bucles, (3) si se encuentra en un bloque condicional, de la probabilidad con la que se ejecute el bloque, y (4) del nivel de anidamiento en expresiones condicionales.

Las operaciones que se contabilizan son: operaciones aritméticas en punto flotante, operaciones aritméticas enteras, acceso a vectores, comparaciones enteras, comparaciones flotantes y

el uso de dos macros existentes en el código para facilitar la representación de los mapas como matrices. La mayoría de estas operaciones contribuyen a la cuenta de instrucciones enteras (IOPs) y a la cuenta de instrucciones en punto flotantes (FLOPs).

Contabilizar únicamente operaciones en punto flotante es simplificar demasiado el modelo de este código. Un algoritmo como éste, diseñado desde el principio para el cálculo eficiente, puede suponer un flujo de instrucciones enteras importante que no debe ser despreciado a priori, a pesar de que la diferencia en el coste entre uno y otro tipo de operaciones pueda ser de varios órdenes de magnitud.

Aparte de las operaciones aritméticas, asimilamos los accesos a vectores a una operación entera (para calcular el desplazamiento), el control de bucles a dos operaciones enteras (comprobación de fin de lazo e incremento del índice), y las comparaciones de los condicionales a una operación entera o en punto flotante según el tipo de variables en la comparación.

Las macros que debemos considerar son:

```
1 #define MS(x,y) = mapa_solucion[(y)*dimx+(x)]
2 #define MU(x,y,u) = mapa_usos[(u)*dimx*dimy+(y)*dimx+(x)]
```

Cada uso de *MS()* se contabilizará como dos operaciones enteras y un acceso a vector. De modo similar, cada uso de *MU()* se contabilizará como 6 operaciones enteras. Estas estructuras se utilizan en todo el código como una ayuda sintáctica al programador, permitiendo simular un acceso por coordenadas *x,y* al estado (*MS*) y a los mapas de aptitud (*MU*), que en realidad se almacenan como vectores unidimensionales.

En la implementación del código los límites de los bucles dependen de valores conocidos a priori (dimensiones de los mapas, número de usos o número de movimientos por temperatura). Por tanto, el número de operaciones con las que contabilizamos lo que contribuye cada línea en un bucle es el mismo que en el caso de que no estuviera en él, multiplicado por el número de iteraciones del bucle.

Cada línea del código dentro de un bloque condicional puede ser ejecutada o no, y la probabilidad de ejecución depende de la cantidad de veces que se evalúe como verdadero. Por tanto, el número de operaciones con las que contabilizamos la contribución de un bloque condicional es el número de operaciones del código del bloque, multiplicado por la probabilidad de

que el condicional se evalúe como verdadero. Se debe tener en cuenta también la contribución del código de la evaluación.

El análisis del código del algoritmo muestra que hay 12 situaciones condicionales diferentes, cada una con su propia probabilidad de ser evaluada como verdadera. Algunas de ellas dependen exclusivamente de factores conocidos a priori y son estáticas en el tiempo. Otras dependen del propio proceso de optimización, de las variables de entrada y de control del algoritmo y del instante temporal, y dada la naturaleza estocástica del proceso no hay forma inmediata de precalcularlas con precisión.

Las probabilidades condicionales encontradas corresponden a los siguientes casos:

- p_1, probabilidad de que el uso asignado a una celda concreta no sea el del uso de máxima aptitud
- p_2, probabilidad de que el uso asignado a una celda concreta no sea el del uso de mínima aptitud
- p_3, probabilidad de que una celda concreta tenga un valor no nulo
- p_4, probabilidad de que una celda concreta esté en la primera fila del mapa
- p_5, probabilidad de que una celda concreta esté en la última fila del mapa
- p_6, probabilidad de que una celda concreta esté en la primera columna del mapa
- p_7, probabilidad de que una celda concreta esté en la última columna del mapa
- p_8, probabilidad de que una celda vecina tenga el mismo uso que una celda dada
- p_9, probabilidad de que dos celdas al azar tengan un valor no nulo y de uso diferente
- p_{10}, probabilidad de aceptación directa del estado actual
- p_{11}, probabilidad de aceptación probabilística del estado actual, después del rechazo en la aceptación directa
- p_{12}, probabilidad de no aceptación probabilística del estado actual, después del rechazo en la aceptación directa

En ocasiones el código utiliza estas probabilidades y sus complementarias para cubrir todo el rango de casos posibles. Por ejemplo, para una probabilidad p_4, el complementario es $(1-p_4)$.

Estas probabilidades son calculadas a partir de la definición, utilizando combinatoria básica (regla de Laplace), para luego comprobar empíricamente que las probabilidades se ajustan a los datos reales medidos. Es un proceso costoso, y tenemos que apoyarnos en simulaciones, ya que a pesar de que, en algunos casos, la diferencia en el significado de dos probabilidades es evidente, las ecuaciones con las que se representan son similares. Esto sucede, por ejemplo, con la diferencia entre las probabilidades p_8 y p_9; en el primer caso, por el propio desarrollo del proceso, sabemos que la celda tiene un valor no nulo, mientras que en el segundo caso no se puede asegurar. Se cumplen las siguientes relaciones:

$$
\begin{gathered}
p_1 = p_2 = 1 - 1/\text{número de usos} \\
p_3 = \text{celdas no nulas}/\text{celdas totales} \\
p_4 = p_5 = 1/\text{filas} \\
p_6 = p_7 = 1/\text{columnas} \\
p_8 = p(\text{celda } i \text{ uso } 0 | \text{celda } j \text{ uso } 0) + p(\text{celda } i \text{ uso } 1 | \text{celda } j \text{ uso } 1) + \cdots = \\
\sum_i \binom{\text{celdas}_i}{2} / \binom{\text{celdas totales}}{2} = \\
\sum_i (\text{celdas}_i \cdot (\text{celdas}_i - 1)) / (\text{celdas totales} \cdot (\text{celdas totales} - 1)) \\
p_9 = \sum_i (\text{celdas}_i \cdot (\text{celdas}_i - 1)) / (\text{celdas totales} \cdot (\text{celdas totales} - 1))
\end{gathered}
\tag{5.7}
$$

donde celdas_i indica el número de celdas asignadas al uso i.

Las probabilidades p_{10}, p_{11} y p_{12}, hacen referencia a las probabilidades de aceptación mostradas antes como p_a, p_b y p_c. Éstas no tienen una expresión analítica inmediata sino que se deben modelar estadísticamente.

A continuación hemos analizado los módulos más importantes y su aportación al modelo en términos de operaciones enteras y en punto flotante. Estos módulos son los encargados del mecanismo de generación de movimientos, evaluación y reevaluación de la función de coste, aceptación y rechazo de transiciones propuestas, enfriamiento y generación del estado inicial. En el último módulo se incorporan las probabilidades dinámicas dependientes de la evolución del SA.

```
calcular_movimiento_aleatorio():
    do
        x1 = random_int(0, columnas)
        y1 = random_int(0, filas)
        x2 = random_int(0, columnas)
        y2 = random_int(0, filas)
        uso1 = MS(x1, y1)
        uso2 = MS(x2, y2)
    while (uso1<0 || uso2<0 || uso1==uso2)
    MS(x1,y1) = TMP_VALUE
    MS(x2,y2) = TMP_VALUE
```

Figura 5.10: Realización de un movimiento aleatorio.

5.4.1. Realización de un movimiento aleatorio

El fragmento de código considerado aparece en la figura 5.10. Consiste en un bucle que contiene cuatro llamadas a una función de coste desconocido, *random_int()*, más dos ejecuciones de las macros que acceden al estado actual, *MS*, y tres comparaciones enteras que definen la condición de terminación del bucle; finalmente incluye un control básico del bucle sin actualización de ningún índice

Analizando el código que usa el compilador para asignar un número de operaciones enteras y en punto flotante a la función *random_int()*, se encuentra que la implementación concreta depende de factores externos como la versión de la librería, utilidades del sistema operativo... Hemos decidimos asignar a esta función *RANDOMINTI* operaciones enteras y *RANDOMINTF* operaciones en punto flotante. Esta función se ejecuta varias veces por cada transición propuesta, quedando estos valores como un factor de escalado para la evolución temporal de FLOPs e IOPs.

Hay un lazo con una condición de finalización que depende del valor concreto de $x1$, $y1$, $x2$, $y2$, $uso1$ y $uso2$, valores que se seleccionan aleatoriamente ($uso < 0$ es la comprobación de que la celda en la posición x,y tiene valor no nulo). Resulta más útil expresar esta condición de terminación como una probabilidad, aprovechando la propiedad de que un lazo con condición de finalización condicional se ejecuta tantas veces como sea necesario hasta que la probabilidad de salida sea igual a 1. Es decir, en un lazo como el anterior, donde la condición

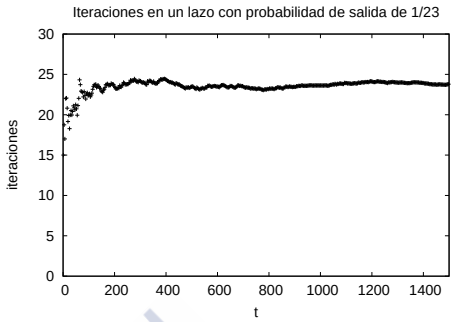

Figura 5.11: Relación entre las iteraciones en un lazo y la probabilidad de terminación p. Cuando el bucle se repite suficiente de veces, el número promedio de veces que se ejecuta el contenido converge a $1/p$. En el código del SA el módulo de generación de movimientos se ejecuta 50 millones de veces para cada temperatura, suficiente para que se cumpla esta propiedad.

de finalización tiene una probabilidad p de evaluarse como verdadera, el código se ejecuta en promedio $1/p$ veces.

Esta probabilidad de terminación del lazo es p_9, cuyo valor ya se ha indicado en la ecuación 5.7.

Por tanto, las instrucciones ejecutadas al llamar al código de generación de movimientos aleatorios son las siguientes *operaciones enteras* (funciones, accesos a macro, operaciones de evaluación de la condición de terminación, actualización):

$$(1/p_9)\cdot(4\cdot\text{RANDOMINTI}+2\cdot 3+3)+2\cdot 3 \tag{5.8}$$

Y las siguientes *operaciones en punto flotante* (ejecución de funciones):

$$(1/p_9)\cdot(4\cdot\text{RANDOMINTF}) \tag{5.9}$$

La condición de terminación del bucle incluye tres condiciones que deben cumplirse simultáneamente. Para poder abordar estas situaciones, a lo largo de todo el código se sustituye cada expresión condicional conteniendo una combinación de varios términos por varias expresiones condicionales de un único término (ver figura 5.12).

En el caso actual la expresión condicional de terminación se ha construido de manera que considere la combinación de las probabilidades de los tres términos y esta sustitución ya no es necesaria.

```
si a y b:
    código

si a:
    si b:
        código
```

```
si a o b:
    código

si a:
    código
en caso contrario:
    si b:
        código
```

Figura 5.12: Relación entre expresiones condicionales complejas y simples.

```
calcular_energia():
    return r1*calcular_energia_t1() +
        r2*calcular_energia_t2()
```

Figura 5.13: Evaluación de la función de coste.

5.4.2. Evaluación de la función de coste

El fragmento relevante se muestra en la figura 5.13. Esta función combina el coste calculado para cada término, que están normalizados, y los reescala por los coeficientes seleccionados por el usuario para el control del proceso de evaluación que son $r1$, coeficiente de ponderación para el término de aptitud, y $r2$, coeficiente de ponderación para el término de compacidad, y que cumplen $0 \leq r1 \leq 1$, $0 \leq r2 \leq 1$ y $r1 + r2 = 1$. El resultado son 3 operaciones en punto flotante.

Analizamos cada uno de los objetivos. El término $t1$ corresponde al cálculo de la aptitud y $t2$, al de la compacidad. Una versión simplificada de la función para el cálculo de la aptitud se muestra en la figura 5.14.

El cálculo de la aptitud inicial se hace una única vez al principio del algoritmo, utilizándose otra función diferente para recalcular y mantener actualizada la aptitud basada en actualizaciones locales. La función actual sirve para calcular la aptitud del estado inicial y al mismo tiempo aprovecha para inicializar los coeficientes de normalización para este término de la función de coste: aptitud máxima, mínima y normalización.

```
calcular_energia_t1():
    aptitud_actual = 0
    aptitud_maxima_global = 0
    aptitud_minima_global = 0
    for i in filas:
        for j in columnas:
            maximo = 0
            minimo = 1
            indice_maximo = -1
            for k in usos:
                valor = MU(j,i,k)
                if valor > maximo:
                    maximo = valor
                    indice_maximo = k
                if valor < minimo:
                    minimo = valor
            aptitud_maxima_global += maximo
            si minimo > -1:
                aptitud_minima_global += minimo
            uso = MS(j,i)
            if uso > -1:
                aptitud_actual += MU(j,i,uso)
    normalizacion_t1 = (aptitud_maxima_global - aptitud_minima_global)
    return (aptitud_maxima_global - aptitud_actual) / normalizacion_t1
```

Figura 5.14: Evaluación del término de aptitud de la función de coste.

Cálculo de la aptitud

El cálculo para la aptitud actual, variable *aptitud_actual*, se encuentra anidado en dos bucles (uno para filas y otro para columnas) y dentro de un condicional asociado a la probabilidad p_3. Así que, considerando sólo el cálculo de aptitud actual, se realizan las operaciones correspondientes a un acceso a la macro *MS*, una comparación entera, un acceso a la macro *MU* con probabilidad p_3 y la operación de actualización mediante suma en punto flotante; todo ello repetido *filas*· *columnas* veces.

Operaciones enteras:

$$\text{filas} \cdot \text{columnas} \cdot (3 + 1 + p_3 \cdot 6) \tag{5.10}$$

Operaciones en punto flotante:

$$\text{filas} \cdot \text{columnas} \cdot (p_3 \cdot 1) \tag{5.11}$$

donde la expresión de la probabilidad p_3 es:

$$p_3 = \text{celdas no nulas}/\text{celdas totales} \tag{5.12}$$

Los probabilidades que rigen en el cálculo de las aptitudes máxima y mínima se asocian a la búsqueda del máximo y del mínimo en un conjunto de valores desordenados. Son las probabilidades p_1 y p_2. Este código se ejecuta al principio del algoritmo y su contribución total es mínima respecto al total de operaciones tanto enteras como en punto flotante. Por tanto, sin pérdida de calidad en el modelo, podemos considerar el peor caso que consiste en que el ordenamiento de los valores es tal que siempre se evalúa alguno de los condicionales como verdaderos (nótese que ambos no pueden ser verdaderos simultáneamente). Por lo que contabilizamos un acceso a la macro *MU* y una comparación flotante, *filas· columnas· usos* veces.

Operaciones enteras:

$$\text{filas} \cdot \text{columnas} \cdot \text{usos} \cdot (6) \tag{5.13}$$

Operaciones en punto flotante:

$$\text{filas} \cdot \text{columnas} \cdot \text{usos} \cdot (1) \tag{5.14}$$

Finalmente se deben considerar las 3 operaciones aritméticas en punto flotante para calcular el normalizador y ofrecer el resultado, y las operaciones de control y actualización de los tres bucles anidados: *2·filas + 2·filas·columnas + 2·filas·columnas·usos*.

El coste total de esta función para el cálculo inicial del término de aptitud viene dado por la siguiente combinación:

Operaciones enteras:

$$\begin{aligned}\text{filas} \cdot \text{columnas} \cdot \text{usos} \cdot (6) + 2 \cdot \text{filas} + 2 \cdot \text{filas} \cdot \text{columnas} + \\ 2 \cdot \text{filas} \cdot \text{columnas} \cdot \text{usos}\end{aligned} \tag{5.15}$$

Operaciones en punto flotante:

$$\text{filas} \cdot \text{columnas} \cdot (p_3 \cdot 1) + \text{filas} \cdot \text{columnas} \cdot \text{usos} \cdot (1) + 3 \tag{5.16}$$

```
calcular_energia_t2():
    perimetro_actual = 0
    perimetro_maximo_global = 0
    perimetro_minimo_global = 0
    for k in usos:
        celdas[k] = 0
        perimetro[k] = 0
    for i in filas:
        for j in columnas:
            k = MS(j,i)
            if k >= 0
                if i > 0
                    if MS(j,i-1) != k
                        perimetro[k]++
                else
                    perimetro[k]++
            ...
            perimetro_maximo_global += 4
            celdas[k]++
    for k in usos:
        perimetro_actual += perimetro[k]
        perimetro_minimo_global += int(sqrt(celdas[k]))
    perimetro_minimo_global *= 4
    normalizacion_t2 = (perimetro_maximo_global - perimetro_minimo_global)
    return (perimetro_actual - perimetro_minimo_global) / normalizacion_t2
```

Figura 5.15: Evaluación del término de compacidad de la función de coste.

Cálculo de la compacidad

El cálculo inicial del término de la compacidad implica la suma de perímetros. De nuevo, esta función sólo se ejecuta una vez al principio del algoritmo, utilizándose otra función diferente para recalcular y mantener actualizados los cambios en el valor de perímetro de cada uso debidos en actualizaciones locales. Esta función sirve para calcular los perímetros de cada uso y su suma total, y al mismo tiempo inicializa los coeficientes de normalización para este término de la función de coste: perímetro máximo, perímetro mínimo y normalización. El código se muestra en la figura 5.15.

Para evaluar los perímetros hay que recorrer cada celda y comparar el uso actual con el uso del vecino al norte, al este, al sur y al oeste (en este caso se utilizan sólo los cuatro vecinos). En el fragmento de la figura 5.15 sólo se muestra la evaluación del vecino del norte para la celda en posición *j,i*. No se incluyen los códigos similares para los otros tres vecinos.

Este condicional se encuentra dentro de dos bucles anidados, uno para las filas y otro para las columnas, y se hacen varias comprobaciones que tienen una probabilidad asociada. En particular, se comprueba si la celda en posición *j,i* es no nula, si la celda está en la primera fila y por tanto no tiene vecino al norte sino el borde del mapa, y en el caso que exista el vecino, si éste es del mismo uso.

En el caso de que la celda sea no nula y esté situada en la primera fila, técnicamente no tiene vecino al norte, pero el borde del mapa incrementa en una unidad el valor de perímetro del uso; en caso de tener vecino al norte, se incrementa el perímetro sólo en el caso de que vecino tenga un uso diferente.

Las probabilidades usadas en este fragmento de código son p_3, probabilidad de que la celda en posición *j,i* tenga valor no nulo, p_4, $1-p_4$ y $1-p_8$. La probabilidad $1-p_8$, complementaria a la probabilidad de que una celda vecina tenga el mismo uso que la celda actual, se calcula como una probabilidad condicional para cada uso. Puesto que no se conoce a qué uso corresponde el valor actualmente en k, hay que calcularlo para todos. De la misma forma, $1-p_4$ es la complementaria a p_4. La expresiones 5.7 muestran cómo obtener estas probabilidades.

Con las operaciones dentro de varios condicionales anidados los casos se tratan como sucesos independientes, y la probabilidad total es el producto de las probabilidades individuales. De esa manera, la probabilidad de que una celda sea útil, no esté en la primera fila y su vecino tenga un uso diferente al suyo es $p_3 \cdot (1-p_4) \cdot (1-p_8)$.

Por tanto, las instrucciones contabilizadas en el caso del cálculo del vecino del norte son el acceso a la macro *MS*, dos comparaciones enteras, una operación de aritmética entera, otro acceso a la macro *MS*, una comparación entera, un acceso a vector y una operación de aritmética entera para el incremento, todo ello escalado por las probabilidades p_3, p_4, $(1-p_4)$ y $(1-p_8)$, según corresponda, y repetido *filas*· *columnas* veces.

Operaciones enteras:

$$\begin{aligned}\text{filas} \cdot \text{columnas} \cdot (2+1+p_3 \cdot (1+(1-p_4) \cdot (1+1+2+(1-p_8) \cdot (1+1))+ \\ p_4 \cdot (1+1)))\end{aligned} \tag{5.17}$$

No hay operaciones en punto flotante.

Hay ecuaciones equivalentes para los otros tres vecinos (sustituyendo p_4 por p_5, p_6 y p_7, respectivamente).

A los cálculos anteriores deberemos sumar las operaciones aritméticas en punto flotante para calcular partes del normalizador y ofrecer el resultado, las operaciones de control y actualización de los bucles en la inicialización y cálculo del normalizador. El valor de normalización se calcula a partir de valores máximos y mínimos ideales.

En total, se obtiene en el término de compacidad los siguientes valores:

Operaciones enteras:

$$\begin{gathered} 4 \cdot \text{usos} + 2 \cdot \text{filas} + 2 \cdot \text{filas} \cdot \text{columnas} + \\ \text{filas} \cdot \text{columnas} \cdot (3 + 2 \cdot p_3 \cdot (1 + (1 - p_4) \cdot (4 + (1 - p_8) \cdot 2) + p_4 \cdot 2) + \\ 2 \cdot p_3 \cdot (1 + (1 - p_6) \cdot (4 + (1 - p_8) \cdot 2) + p_6 \cdot 2) + 3) + 2 \cdot \text{usos} + \\ \text{usos} \cdot (6 + \text{SQRTI}) \end{gathered} \tag{5.18}$$

Operaciones en punto flotante:

$$\text{usos} \cdot \text{SQRTF} + 1 \tag{5.19}$$

donde consideramos $p_4 = p_5$, $p_6 = p_7$, y hemos utilizado *SQRTI* como el número equivalente de operaciones enteras dentro la función de coste desconocido *sqrt()*, y *SQRTF* como el de operaciones en punto flotante dentro de la misma función. Independientemente del valor exacto, que podemos consultar en la librería que utilice el compilador, esta función se ejecuta una única vez al principio del algoritmo y las aportaciones de *SQRTx* son testimoniales sobre el total.

Para la evaluación completa, se obtienen los valores:

Operaciones enteras:

$$\begin{gathered} \text{filas} \cdot \text{columnas} \cdot (3 + 1 + p_3 \cdot 6) + \text{filas} \cdot \text{columnas} \cdot \text{usos} \cdot (6) + 2 \cdot \text{filas} + \\ 2 \cdot \text{filas} \cdot \text{columnas} + 2 \cdot \text{filas} \cdot \text{columnas} \cdot \text{usos} + 4 \cdot \text{usos} + 2 \cdot \text{filas} \\ + 2 \cdot \text{filas} \cdot \text{columnas} + \text{filas} \cdot \text{columnas} \cdot (3 + 2 \cdot p_3 \cdot (1 + \\ (1 - p_4) \cdot (4 + (1 - p_8) \cdot 2) + p_4 \cdot 2) + 2 \cdot p_3 \cdot (1 + \\ (1 - p_6) \cdot (4 + (1 - p_8) \cdot 2) + p_6 \cdot 2) + 3) + 2 \cdot \text{usos} + \text{usos} \cdot (6 + \\ \text{SQRTI}) \end{gathered} \tag{5.20}$$

Operaciones en punto flotante:

$$\begin{gathered} 3 + \text{filas} \cdot \text{columnas} \cdot (p_3 \cdot 1) + \text{filas} \cdot \text{columnas} \cdot \text{usos} \cdot (1) + 3 + \\ \text{usos} \cdot \text{SQRTF} + 1 \end{gathered} \tag{5.21}$$

```
recalcular_energia():
    return r1*recalcular_energia_t1() +
        r2*recalcular_energia_t2()
```

Figura 5.16: Reevaluación de la función de coste.

```
recalcular_energia_t1():
    aptitud_actual = aptitud_actual
        - MU(x1,y1,uso1)
        - MU(x2,y2,uso2)
        + MU(x1,y1,uso2)
        + MU(x2,y2,uso1)
    return (aptitud_maxima_global - aptitud_actual) / normalizacion_t1
```

Figura 5.17: Reevaluación del término de aptitud de la función de coste.

5.4.3. Reevaluación de la función de coste

El recálculo del coste se muestra en la figura 5.16. Durante la ejecución del algoritmo, para cada transición propuesta, se reevalúa el coste llamando a esta función. El resultado vuelven a ser 3 operaciones de aritmética en punto flotante.

Lo interesante es el modo de reevaluar cada uno de los objetivos. Al igual que en la sección 5.4.2, término $t1$ corresponde al recálculo de la aptitud y $t2$, al de la compacidad. Ocultando detalles irrelevantes de la implementación, el primer término se muestra en la figura 5.17 y el segundo término en la figura 5.18.

Se observa que, después del intercambio de usos entre las celdas *x1,y1* y *x2,y2*, la aptitud actual se recalcula a partir del valor global que se ha almacenado durante la función *calcular_energia_t1()* al principio del algoritmo, eliminando la contribución de aptitud que tenían los usos originales y añadiendo la que tienen los nuevos usos.

En el cálculo de instrucciones consideramos las operaciones enteras relacionadas con el acceso a la macro *MU*, las operaciones aritméticas en punto flotante para recalcular la aptitud y normalizar.

Hay $4 \cdot 6$ operaciones enteras, y $4 + 2$ operaciones en punto flotante.

```
recalcular_energia_t2():
    a = 0
    n = 0
    if y1 > 0
        k = MS(x1,y1-1)
        if k == uso1
            a++
        if k == uso2
            n++
    ...
    perimetro[uso1] += (a*2 - 4)
    perimetro[uso2] += ((4 - n)*2 - 4)
    perimetro_actual += (a*2-4) + ((4-n)*2-4)
    a = 0
    n = 0
    if y2 > 0
        k = MS(x2,y2-1)
        if k == uso2
            a++
        if k == uso1
            n++
    ...
    perimetro[uso2] += (a*2 - 4)
    perimetro[uso1] += ((4 - n)*2 - 4)
    perimetro_actual += (a*2-4) + ((4-n)*2-4)
    return (perimetro_actual - perimetro_minimo_global) / normalizacion_t2
```

Figura 5.18: Reevaluación del término de compacidad de la función de coste.

En esta implementación, las sucesivas operaciones aritméticas con los valores de los mapas de aptitud (macro *MU*) pueden introducir poco a poco errores en el valor de aptitud, que se calcula una y otra vez, debido a redondeos y errores de representación para números pequeños. Una posibilidad para evitarlo es trabajar con tipos de datos de diferentes rangos, usando uno mayor para almacenar la aptitud, de manera que este dato soporte las operaciones con los valores de los mapas de aptitud sin pérdida de precisión. La alternativa utilizada en RULES es trabajar con el mismo tipo de datos, pero limitar el número de decimales de los datos contenidos en los mapas de aptitud.

Se observa que, después del intercambio de usos entre las celdas *x1,y1* y *x2,y2*, los valores de perímetro global de los usos se modifican porque se modifican las fronteras. El proceso se hace en dos pasos. Primero se calcula el cambio en el valor del perímetro alrededor de una celda. Luego se repite para la segunda.

Para calcular el cambio de perímetro alrededor de la primera celda hay que contabilizar el número de vecinos que tienen el uso original *uso*1 y los que tienen el nuevo uso *uso*2. El valor se almacena en las variables a y n. Cuando hay una coincidencia con el uso antiguo, al aplicar el nuevo uso aparece una frontera entre celda y vecino, y se contabiliza como perímetro dos veces: como perímetro de la celda y como perímetro del vecino. Cuando hay una coincidencia con el uso nuevo, al aplicar el uso nuevo desaparece esa misma frontera entre celda y vecino ya que ahora comparten el mismo uso, y, de nuevo, se elimina dos veces: uno por la celda, y otro por el vecino.

Es decir, al sustituir en la primera celda, la asignación al *uso*1 por asignación al *uso*2, se observan los siguientes cambios:

$$\begin{aligned} &\text{perimetro}[\text{uso1}] += (a \cdot 2 - 4) \\ &\text{perimetro}[\text{uso2}] += ((4-n) \cdot 2 - 4) \\ &\text{perimetro\_actual} += (a \cdot 2 - 4) + ((4-n) \cdot 2 - 4) \end{aligned} \tag{5.22}$$

La figura 5.18 sólo muestra la evaluación del vecino del norte para la celda en posición *x1,y1*. Falta las comprobaciones para los otros tres vecinos.

La comparación actual no está dentro de ningún bucle, y hace varias comprobaciones con una probabilidad asociada conocida: comprueba si la celda en posición *x1,y1* no está en la primera fila y por tanto se puede considerar el vecino al norte, y comprueba si el uso asignado al vecino coincide con el uso antiguo o con el nuevo.

Las probabilidades usadas en este fragmento de código son $1 - p_4$ y p_8. Cuando extendemos el código para considerar al resto de vecinos, añadimos las probabilidades $1 - p_5$, $1 - p_6$ y $1 - p_7$.

En resumen, la primera mitad de la función de recálculo de la compacidad, correspondiente a las modificaciones de perímetro actual debidas al cambio de asignación de la celda en posición *x1,y1* del uso *uso*1 al uso *uso*2, y considerando los cuatro vecinos, realiza las siguientes operaciones:

Operaciones enteras (comparación entera, aritmética entera más acceso a macro MS, comparación entera y actualización, todo ello con distintas probabilidades asociadas a cada bloque

condicional, y varias operaciones aritméticas para la actualización del perímetro):

$$\begin{array}{c} 1+2\cdot((1-p_4)\cdot(3+2\cdot(1+p_8\cdot 1)))+ \\ 2\cdot((1-p_6)\cdot(3+2\cdot(1+p_8\cdot 1)))+4+5+7 \end{array} \tag{5.23}$$

No tiene operaciones en punto flotante.

La segunda mitad de la función de recálculo de la compacidad, correspondiente a las modificaciones de perímetro actual debidas al cambio de asignación de la celda en la posición *x2,y2* del uso *uso*2 al uso *uso*1, tiene una expresión idéntica. Para el cálculo global del número de operaciones hace falta considerar también la normalización, que es el único punto en que aparecen operaciones en punto flotante.

Operaciones enteras:

$$\begin{array}{c} 2\cdot(1+2\cdot((1-p_4)\cdot(3+2\cdot(1+p_8*1)))+ \\ 2\cdot((1-p_6)\cdot(3+2\cdot(1+p_8\cdot 1)))+4+5+7) \end{array} \tag{5.24}$$

Hay 2 operaciones en punto flotante.

En total, se obtienen, para las operaciones enteras:

$$\begin{array}{c} 24+2\cdot(1+2\cdot((1-p_4)\cdot(3+2\cdot(1+p_8\cdot 1)))+ \\ 2\cdot((1-p_6)\cdot(3+2\cdot(1+p_8\cdot 1)))+4+5+7) \end{array} \tag{5.25}$$

Y las operaciones en punto flotante son $6+2$.

5.4.4. Aceptación de la transición

El proceso de aceptación de una transición no supone operaciones de cálculo, sino que lleva asociada una copia, dentro de variables globales, de distintos valores que se han calculado en la reevaluación de la función de coste con la transición propuesta. Se muestra en la figura 5.19.

Únicamente consideramos las operaciones enteras resultado de acceder a las macros. Son $2\cdot 2$ operaciones enteras, y no hay operaciones en punto flotante.

```
aceptar_movimiento():
    MS(x1,y1) = uso2
    MS(x2,y2) = uso1
    aceptar_movimiento_t1()
    aceptar_movimiento_t2()

aceptar_movimiento_t1():
    nada

aceptar_movimiento_t2():
    nada
```

Figura 5.19: Aceptación de la transición.

```
rechazar_movimiento():
    MS(x1,y1) = uso1
    MS(x2,y2) = uso2
    energia = energia_original
    rechazar_movimiento_t1()
    rechazar_movimiento_t2()

rechazar_movimiento_t1():
    aptitud_actual = aptitud_actual_original

rechazar_movimiento_t2():
    perimetro_actual = perimetro_actual_original
    perimetro[uso1] = perimetro_uso1_original
    perimetro[uso2] = perimetro_uso2_original
```

Figura 5.20: Rechazo de la transición.

5.4.5. Rechazo de la transición

En este caso no hay cálculo asociado, sino sólo recuperación de las variables originales sobre las globales. Se muestra en la figura 5.20. Y de nuevo las operaciones consideradas son las correspondientes a las llamadas para la actualización de cada término de la función de coste.

Las operaciones enteras son $2 \cdot 2 + 2$, y no hay operaciones en punto flotante.

En el código puede observarse que, al proponer una transición se almacena una copia del estado actual (asignación de celdas a usos) y de algunas variables derivadas (coste actual, evaluación actual de cada uno de los términos de la función de coste), que son recuperadas si

finalmente la transición se rechaza. En el caso de que la transición sea aceptada, los nuevos valores calculados ya están copiados a las variables que representan el estado actual. Es por eso las funciones de *aceptar_movimiento_t1()* y *aceptar_movimiento_t2()* están vacías mientras que *rechazar_movimiento_t1()* y *rechazar_movimiento_t2()* no lo están.

5.4.6. Generación del estado inicial aleatorio

Al principio del programa, después de cargar y revisar los mapas de aptitud, la intersección de todos ellos proporciona una máscara del estado inicial que se accede desde la macro *MS*. Es un raster con valores enteros 0 o -1, que indican si la celda en esa posición contendrá en el futuro inmediato una asignación a uso o si la celda contiene el valor nulo. Ignoramos el proceso concreto de lectura de los mapas e intersección para crear esta máscara, ya que durante todo el análisis estamos ignorando las operaciones de entrada/salida.

La generación inicial más sencilla sitúa consecutivamente de izquierda a derecha y de arriba abajo las celdas de cada uso. Primero todas las celdas del uso 0, luego las del uso 1, etc. hasta finalizar. En este momento todas las celdas no nulas tienen un uso asignado, y el número de celdas por uso coincide con el número solicitado.

A continuación se reordenan las celdas aleatoriamente, intercambiando cada una por una pareja aleatoria. Para aumentar la aleatoriedad del resultado, se rodea el proceso básico de un bucle externo para repetir el proceso de reordenamiento varias veces. El código utilizado se muestra en la figura 5.21, y su análisis muestra las siguientes operaciones:

Operaciones enteras:

$$\begin{gathered} 2\cdot \text{filas} + 2\cdot \text{filas}\cdot \text{columnas} + \text{filas}\cdot \text{columnas}\cdot (2+1+p_3\cdot (2+1)+ \\ \text{celdas\_utiles}\cdot (3)) + 2\cdot \text{reordenamientos}+ \\ 2\cdot \text{reordenamientos}\cdot \text{filas} + 2\cdot \text{reordenamientos}\cdot \text{filas}\cdot \text{columnas}+ \\ \text{reordenamientos}\cdot \text{filas}\cdot \text{columnas}\cdot (2+1+p_3\cdot (2\cdot \text{RANDOMINTI}+ \\ 2+1+p_3\cdot (2\cdot 2))) \end{gathered} \tag{5.26}$$

Operaciones en punto flotante:

$$\text{reordenamientos}\cdot \text{filas}\cdot \text{columnas}\cdot p_3\cdot 2\cdot \text{RANDOMINTF} \tag{5.27}$$

```
estado_inicial_aleatorio():
    k = 0
    nk = 0
    for i in filas:
        for j in columnas:
            if MS(j,i) > -1:
                if nk > celdas[k]:
                    k++
                    nk = 0
                MS(j, i) = k
                nk++

    for r in reordenamientos:
        for i in filas:
            for j in columnas:
                if MS(j, i) > -1:
                    ni = random_int(i, filas)
                    nj = random_int(j, columnas)
                    if MS(nj, ni) > -1:
                        tmp = MS(j, i)
                        MS(j, i) = MS(nj, ni)
                        MS(nj, ni) = tmp
```

Figura 5.21: Generación del estado inicial aleatorio.

Hemos reutilizado los valores *RANDOMINTI* y *RANDOMINTF* para representar los valores del número de operaciones enteras y flotantes dentro de la función *random_int()*, tal como estaban definidas en el módulo de generación de movimientos.

5.4.7. Evolución probabilística del proceso de optimización

Las probabilidades usadas para modelar los módulos son estáticas, y se pueden calcular atendiendo a los parámetros que definen el tamaño del problema: número de filas, número de columnas, número de celdas no nulas, número de usos considerados, número de celdas objetivo por cada uso. Esto permite eliminar la necesidad de tratar con la distribución concreta de valores en los mapas de aptitud o con la situación inicial de asignaciones de usos en el estado inicial. Añadimos a este conjunto de parámetros el número de movimientos en cada temperatura y el número máximo de temperaturas definido por el criterio de parada.

Sin embargo falta considerar las probabilidades propias del proceso de optimización p_{10}, p_{11} y p_{12}. Estas probabilidades son dinámicas: dependen de las condiciones iniciales, pero

evolucionan en el tiempo, dándole al algoritmo de SA una ventaja competitiva respecto a los procesos de búsqueda local básicos que siguen un comportamiento prefijado.

La probabilidad p_{10} está relacionada con el proceso de aceptación directa, que es esencial en el proceso de optimización: se acepta cualquier iteración que mejore el coste del estado actual. El valor de p_{10} no es estático, sino que la posibilidad de terminar en un estado con coste inferior al actual al realizar una perturbación disminuye a medida que el algoritmo avanza. Al principio, como el sistema arranca desde un estado inicial aleatorio, encontrar mejores estados es muy probable. Pero en etapas posteriores, según el proceso converge hacia el óptimo, cada vez hay menos probabilidades.

La probabilidad p_{11} y su complementaria p_{12} están relacionadas con el proceso de aceptación probabilístico de Metropolis. Junto con las aceptaciones directas, se aceptan algunas transiciones que aumentan el coste del estado actual con una probabilidad que depende del coste del estado y de la temperatura actual del sistema. La posibilidad de aceptar estas transacciones con un coste superior al actual también disminuye a medida que el algoritmo avanza. El efecto del parámetro temperatura hace que al principio, con altas temperaturas, se acepten muchas de estas iteraciones. En etapas posteriores cada vez se aceptan menos según disminuye la temperatura.

En este apartado modelamos estas probabilidades. Inicialmente, consideramos que dependen de las condiciones iniciales, parámetros de control y del tiempo:

$$\begin{aligned} p_{10} &\sim p_{10}(\text{condiciones iniciales}, \text{control}, \text{tiempo}) \\ p_{11} &\sim p_{11}(\text{condiciones iniciales}, \text{control}, \text{tiempo}) \end{aligned} \tag{5.28}$$

Las probabilidades p_{10}, p_{11} y p_{12} determinan las tres vías de ejecución del código, tal y como se mostró en la figura 5.9.

Hasta el momento se han calculado las operaciones enteras y en punto flotante para los distintos módulos, y ahora se unen. El esquema completo se muestra en la figura 5.22.

Como se puede ver en esta figura, se ejecuta una única vez el conjunto de instrucciones producidas por el módulo de creación del estado inicial aleatorio. También el módulo para calcular la energía del estado inicial a partir de la evaluación de cada uno de los términos de la función de coste.

```
algoritmo():
    estado_inicial_aleatorio()
    energia = evaluar_coste()
    for t in temperaturas:
        for i in movimientos_por_temperatura:
            calcular_movimiento_aleatorio()
            nueva_energia = reevaluar_coste()
            if nueva_energia <= energia:
                aceptar_movimiento()
            else
                p = exp((energia - nueva_energia)/t);
                if rand() <= p:
                    aceptar_movimiento()
                else:
                    rechazar_movimiento()
```

Figura 5.22: Esquema del algoritmo y los módulos.

A continuación hay un lazo exterior que recorre todas las temperaturas del calendario de enfriamiento. En la implementación cada temperatura se genera a partir de la anterior, llamando a la función *nueva_temperatura(t)*, que implica una única operación aritmética en punto flotante. En el pseudocódigo anterior se ha representado como un bucle precalculado de temperaturas. Para cada temperatura hay un nuevo lazo anidado que permite generar un número fijo de movimientos. Dentro de éste se proponen las transiciones, se evalúa la energía de los nuevos estados, y se someten a los tests de aceptación. Con una probabilidad p_{10}, el nuevo estado se acepta. Con una probabilidad $1 - p_{10}$, el estado se somete al test de aceptación de Metropolis. Que se supera con una probabilidad p_{11}, y se rechaza con probabilidad p_{12}.

En conjunto, las instrucciones consideradas son las que aparecen en la figura 5.23.

Únicamente se dispone de expresiones analíticas para p_a, p_b y p_c (p_{10}, $(1 - p_{10}) \cdot p_{11}$ y $(1 - p_{10}) \cdot p_{12}$, respectivamente) en casos muy particulares, donde el generador de transiciones cumple una serie de limitaciones [139]. Para el caso general, se necesita recurrir a una evaluación estadística. Para ello se ejecuta el algoritmo múltiples veces y con un rango variado de parámetros de entrada y de control, y del análisis post mortem de los procesos se extrae la información sobre la evolución de las probabilidades.

En estas ejecuciones tenemos que limitar el número de parámetros. Asumimos que es la combinación de parámetros de entrada la que define un problema particular. Así, podemos dejar

```
operaciones_estado_inicial_aleatorio +
operaciones_evaluar_coste +
temperaturas*movimientos_por_temperatura*(
    operaciones_calcular_movimiento_aleatorio +
    operaciones_reevaluar_coste +
    operaciones_test_aceptacion_directa +
    p10*operaciones_aceptar_movimiento +
    (1-p10)*(operaciones_test_aceptacion_metropolis +
        p11*operaciones_aceptar_movimiento +
        p12*operaciones_rechazar_movimiento)
    )
)
```

Figura 5.23: Esquema de módulos para la contabilidad de instrucciones.

fijos el tamaño del mapa, el número de celdas con valor no nulo y sus posiciones, el número de usos, el número de celdas solicitadas por cada uso y la distribución de los mapas de aptitud de los usos.

Para simplificar fijamos también dos parámetros de control: el número de temperaturas en el calendario de enfriamiento y el número de movimientos propuestos a cada temperatura, y utilizamos como parámetros de control libres la temperatura inicial, la constante de enfriamiento del calendario de enfriamiento lineal, y los coeficientes de cada término de la función de coste. La evolución temporal viene dada por el número de etapa de temperatura, que es el último parámetro libre.

A modo de ejemplo, resolvemos el problema definido por los siguientes parámetros de entrada:

- 2695 filas
- 2846 columnas
- 4315411 celdas no nulas (56 % del total)
- 13 usos:

  eucalipto: solicitadas 193675 celdas
  forrvp: 695875 celdas
  frondosas: 690175 celdas
  frutales: 5600 celdas
  hortalizas: 363250 celdas

maiz: 757818 celdas
otrcereales: 3550 celdas
otrforr: 63125 celdas
pastizales: 115725 celdas
patata: 47700 celdas
prados: 774668 celdas
resinosas: 554025 celdas
trigo: 50225 celdas

- la distribución concreta de los 13 mapas de aptitud
- 500000000 posibles movimientos por cada temperatura
- 500 temperaturas

El estado inicial es aleatorio y diferente en cada ejecución. Esto permite no particularizar para ningún caso concreto. Pero, al mismo tiempo, facilita iniciar el algoritmo en un máximo de entropía por la forma de generarlo, que es a la vez un estado conocido y suficientemente genérico. Esto es porque la evolución temporal del SA, como se ha indicado en la sección 5.3, es poco dependiente del estado aleatorio inicial concreto.

Hacemos un análisis de la evolución temporal del algoritmo de SA para este problema, centrándonos en las probabilidades de aceptación directa, aceptación por el criterio de Metropolis y rechazo, para un rango de parámetros de control:

- 41 valores para la temperatura inicial, entre 0.10 y 50.00
- 5 valores de constante de enfriamiento, entre 0.95 y 0.99
- 20 combinaciones para los coeficientes de balanceo de la función de coste, entre (0.00, 1.00) y (1.00, 0.00)
- 500 medidas temporales, desde la etapa 0 a la etapa 499

En total, resultan 4100 curvas (cerca de dos millones de puntos).

A partir de los valores medidos durante las ejecuciones construimos un modelo estadístico del comportamiento temporal de p_a, p_b y p_c para parámetros de control arbitrarios que seleccione el usuario, siempre que se localicen dentro del rango utilizado para el modelo.

Nótese que este análisis es sólo válido para esta implementación concreta del SA, con la misma función de coste, el mismo mecanismo de generación de movimientos, los mismos parámetros de entrada señalados y con un estado inicial aleatorio. No es un análisis general del

proceso de optimización por SA, aunque es representativo, pero permite aproximar cualquier algoritmo de SA y extraer conclusiones.

Como aproximación al problema, visualizamos el comportamiento que presentan estas curvas en función de los parámetros de control. Representamos la evolución temporal de las probabilidades, donde el tiempo está representado por el número de etapa del calendario de enfriamiento, cuando fijamos dos de los parámetros de control y dejamos libre el tercero. Los parámetros son la temperatura inicial, la constante de enfriamiento y los coeficientes de los términos de la función de coste (donde consideramos los dos coeficientes como un único parámetro dado que su suma es 1).

En la figura 5.24 se representa p_a a través del número de aceptaciones directas, de un total de 50 millones de posibles movimientos por temperatura, para todos los valores disponibles de la temperatura inicial (41 valores). A continuación fijamos la constante de enfriamiento al valor, por ejemplo, de $ct = 0{,}8$, y una combinación concreta de coeficientes para los términos de la función de coste, por ejemplo, 0.8 para el término de aptitud y 0.2 para el de compacidad ($ru = 0{,}8$).

En las curvas de la figura 5.24 se puede observar que el comportamiento presenta un esquema característico. El aspecto general es el de una función exponencialmente decreciente, con una primera etapa de caída lenta, más prolongada cuanto mayor es la temperatura inicial del proceso. Eso quiere decir que en el SA, cuanto mayor sea la temperatura en las primeras etapas más tiempo se emplea en una exploración alrededor del estado inicial y más tiempo tarda el algoritmo en comenzar su convergencia hacia la solución. Según disminuye la temperatura inicial, las gráficas se desplazan hacia la izquierda mientras mantienen un aspecto similar. Eso indica que una temperatura inicial más baja no permite explorar el mismo rango de posibilidades en las primeras etapas y rápidamente comienza su convergencia hacia la solución desde la mejor situación que ha podido localizar en las etapas iniciales.

Las representaciones del número de aceptaciones directas tomando como variable libre la constante de enfriamiento o la combinación de coeficientes en la función de coste, presenta características similares tal como se puede observar en la figura 5.25. Nótese que el proceso de aceptaciones está afectado por los valores iniciales de cada uno de los parámetros de control.

En este punto del análisis debemos utilizar alguna herramienta para generar automáticamente un modelo para esta curva y la de p_b (la de p_c se calcula por exclusión). Para ello disponemos

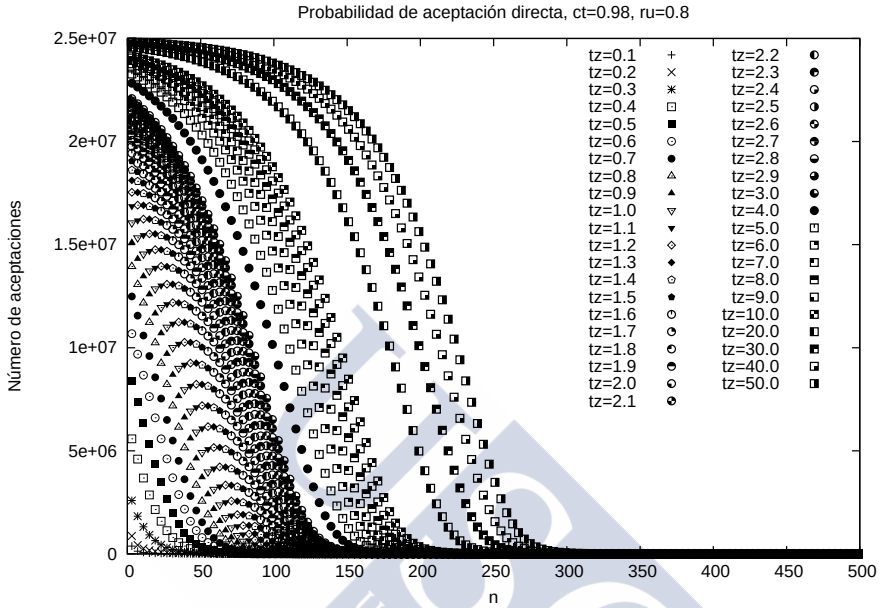

Figura 5.24: Aceptación directa en función de la temperatura inicial, manteniendo fijos la constante de enfriamiento y los coeficientes de la función de coste.

de más de dos millones de puntos de ajuste y cuatro variables independientes: temperatura inicial, constante de enfriamiento, coeficientes de la función de coste y el número de etapa del calendario de enfriamiento como parámetro temporal.

Para el caso particular de esta implementación del SA podemos continuar y realizar una simplificación más que facilita encontrar el modelo final, ya que tres de los cuatro parámetros de control están relacionados entre sí a través del calendario de enfriamiento: la temperatura inicial, la constante de enfriamiento y el número de etapa de enfriamiento. Además, la temperatura viene dada por la secuencia $\{t_0, t_1 = t \cdot ct, t_2 = t_1 \cdot ct, t_3 = t_2 \cdot ct \ldots\}$. Es decir, para la etapa n, se tiene:

$$t(n) = t_0 \cdot ct^n \tag{5.29}$$

Representando las mismas gráficas, utilizando como parámetro temporal la temperatura de la etapa en lugar del número de etapa, aparecen dos diferencias fundamentales que pueden observarse en las gráficas 5.26 y 5.27. Por un lado, las curvas hay que leerlas ahora de derecha

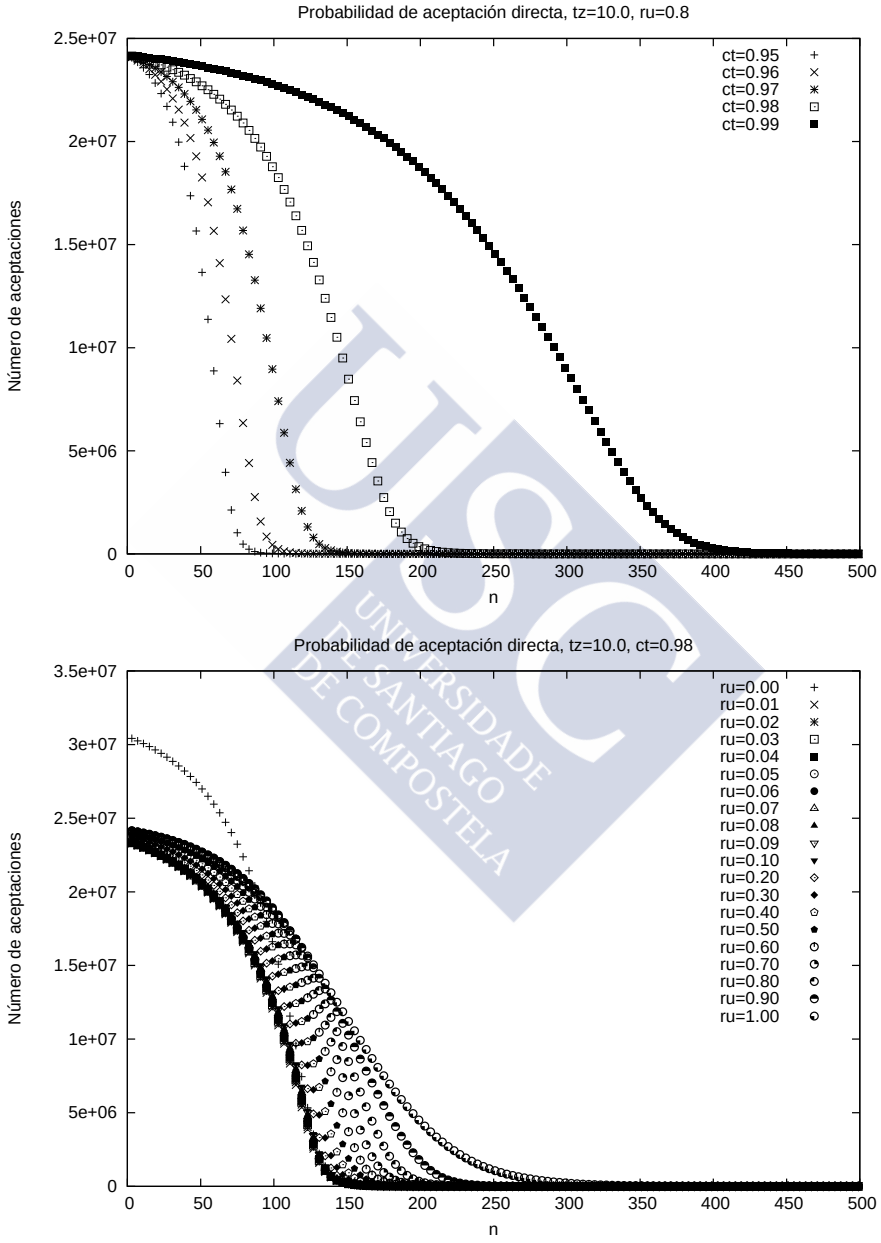

Figura 5.25: Aceptación directa en función de la constante de enfriamiento, manteniendo fijos los otros parámetros, y aceptación directa en función de los coeficientes de la función de coste, manteniendo fijos los otros.

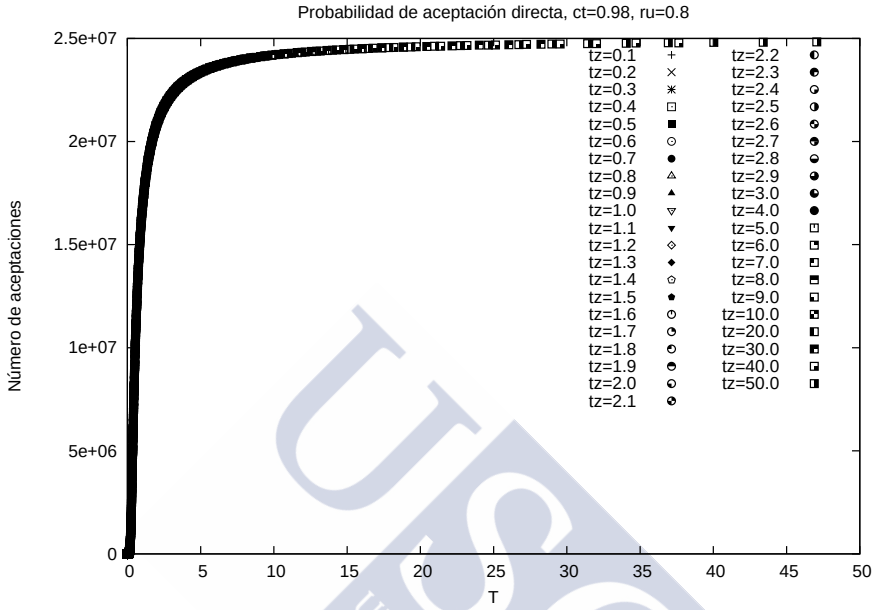

Figura 5.26: Aceptación directa en función de la temperatura inicial, manteniendo fijos los otros parámetros, usando la variable temperatura.

a izquierda, porque según avanza el proceso la temperatura disminuye. Por otro, que las curvas en función de la temperatura inicial o constante de enfriamiento prácticamente se superponen, pero no así la que está en función de los coeficientes de la función de coste.

Esto indica que la evolución del modelo, cuando la variable temporal es la temperatura, depende fundamentalmente del valor del parámetro de control *ru*, de los coeficientes de la función de coste, y, muy poco, de la temperatura inicial, *tz*, o de la constante de enfriamiento, *ct*. Podemos centrarnos inicialmente en modelar esta dependencia. Por lo tanto, simplificamos el modelo de cuatro variables independientes a sólo dos: los coeficientes de la función de coste y temperatura de la etapa.

Si bien todas esas curvas tienen un esquema similar y el comportamiento básico es que la probabilidad aumenta cuando disminuye el parámetro *ru*, hay algunas que se escapan del caso general (ver figura 5.28). Corresponden a la familia de curvas con $ru = 0{,}0$ y presentan, por comparación con el resto de valores de *ru*, más aceptaciones. Debido a los valores máximos y

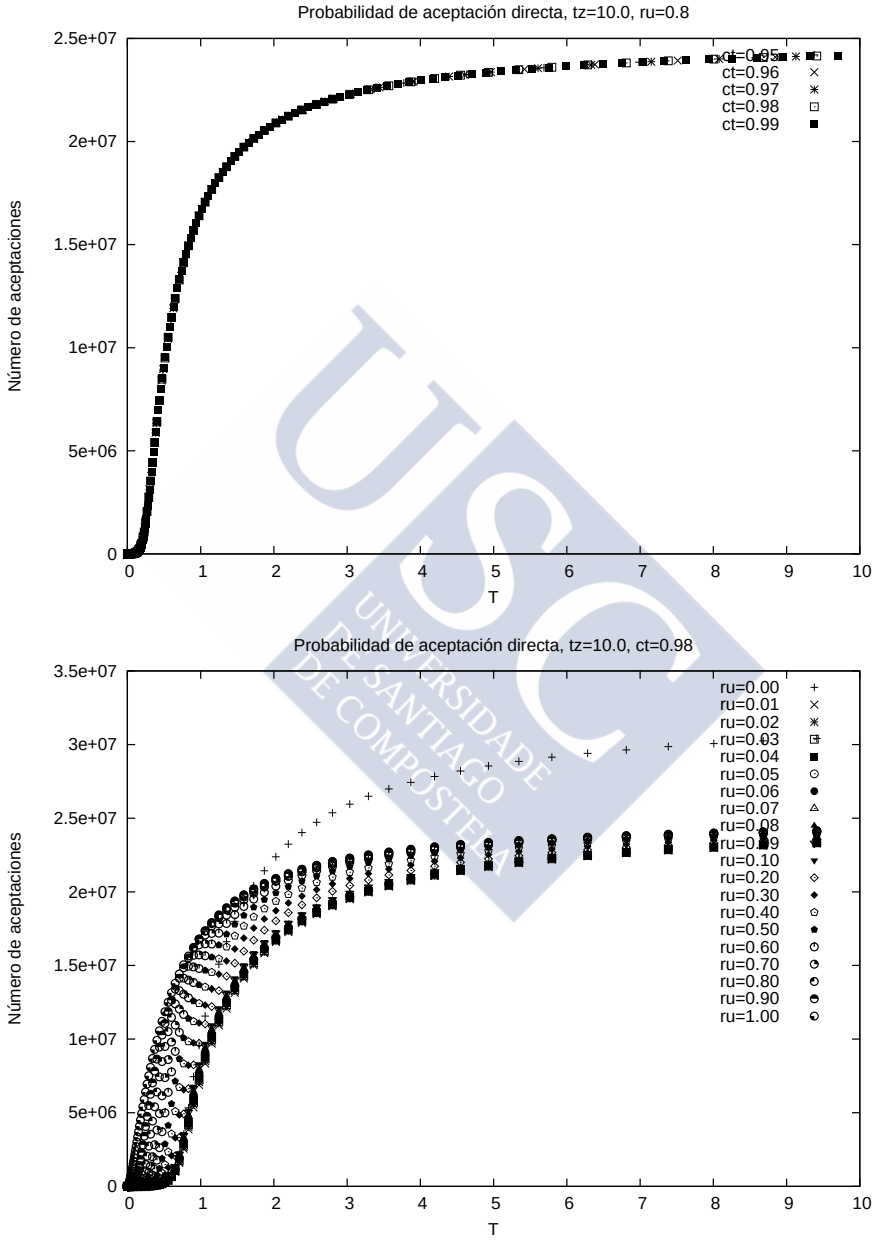

Figura 5.27: Aceptación directa en función de la constante de enfriamiento o los coeficientes de la función de coste, manteniendo fijos los otros parámetros, usando la variable temperatura.

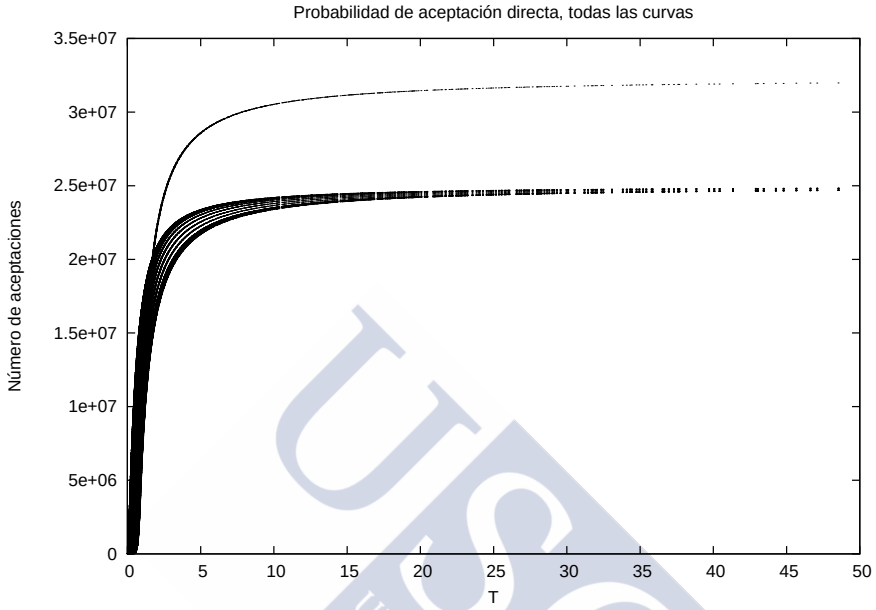

Figura 5.28: Todas las medidas de aceptación directa, representadas en la variable temperatura. Se agrupan en familias con un valor común del parámetro *ru*.

mínimos utilizados en la normalización, las diferencias en los valores de los términos de aptitud y de compacidad cuando se realiza un movimiento tienen rangos diferentes, y el cambio en el valor de la compacidad queda ligeramente enmascarado por el cambio en el valor de la aptitud.

Cuando nos centramos sólo en el término de compacidad, encontramos que en las primeras etapas, donde las posiciones iniciales son aleatorias, muchos movimientos mejoran o mantienen el valor de compacidad. Es más sencillo cumplir sólo con uno de los términos de la función de coste que con los dos, pero, en este caso, es la normalización del término de compacidad, por la diferencia de rangos en que trabajan ambos términos, el que establece el número de aceptaciones.

En cualquier caso, es un efecto que debemos incorporar al modelo. La forma más inmediata para ello es considerar de forma independiente los casos con $ru = 0{,}0$ (donde sólo se evalúa

compacidad). Y, por simetría, también considerar de manera especial el caso $ru = 1{,}0$ (donde sólo se evalúa aptitud).

Analicemos más en detalle el caso de una familia de curvas cualquiera. La familia con $ru = 0{,}8$ se muestra en la figura 5.29. A bajas temperaturas destacan algunos puntos que se alejan de la tendencia general de la familia de curvas. Corresponden a valores de aceptaciones directas en las primeras etapas de una ejecución que están ligeramente por encima de los valores esperados, presentando una estructura visual en forma de dientes de sierra. El punto más elevado corresponde a la temperatura inicial de la curva, y luego los puntos obtenidos se mueven hacia la izquierda con varias pendientes diferentes, cada una corresponde a una constante de enfriamiento distinta. Se observa también que estos picos son más acusados cuando las temperaturas iniciales son más bajas, y que sus efectos sólo se aprecian durante unas pocas temperaturas.

Por tanto, podemos repetir la simplificación anterior y modelar las primeras temperaturas de cada curva independientemente del resto de temperaturas.

En resumen, vamos a modelar la evolución temporal de la probabilidad de aceptación directa con varias ecuaciones:

- Caso ru = 0.0
  etapa n = 0, etapa n = 1, etapa n = 2 y el resto de las etapas
- Caso ru = 1.0
  etapa n = 0, etapa n = 1, etapa n = 2, y el resto
- Otros valores de ru
  etapa n = 0, etapa n = 1, etapa n = 2, y el resto

Hemos empleado distintas herramientas para encontrar el mejor ajuste de estas ecuaciones, ahora con sólo dos variables independientes (ru y t) y con un número de puntos de ajuste que depende de la ecuación que buscamos (en nuestro ejemplo, desde 205 puntos para el caso ru=0.0, n=0, hasta más de 1.8 millones de puntos para el caso general). Así, se ha hecho uso de una variedad de productos software (*MathWorks MATLAB* [163], *GNU Octave* [214], *Systat TableCurve* [84]) para identificar la forma general de estas ecuaciones, los términos que podemos esperar en ellas y su dependencia con las variables, y hemos combinado estas expectativas a través de la herramienta TIA [162] para llegar a un resultado final.

TIA (*Tools for Instrumentation and Analysis*) es un entorno de análisis diseñado para obtener de forma sencilla modelos analíticos precisos del rendimiento de aplicaciones paralelas. A

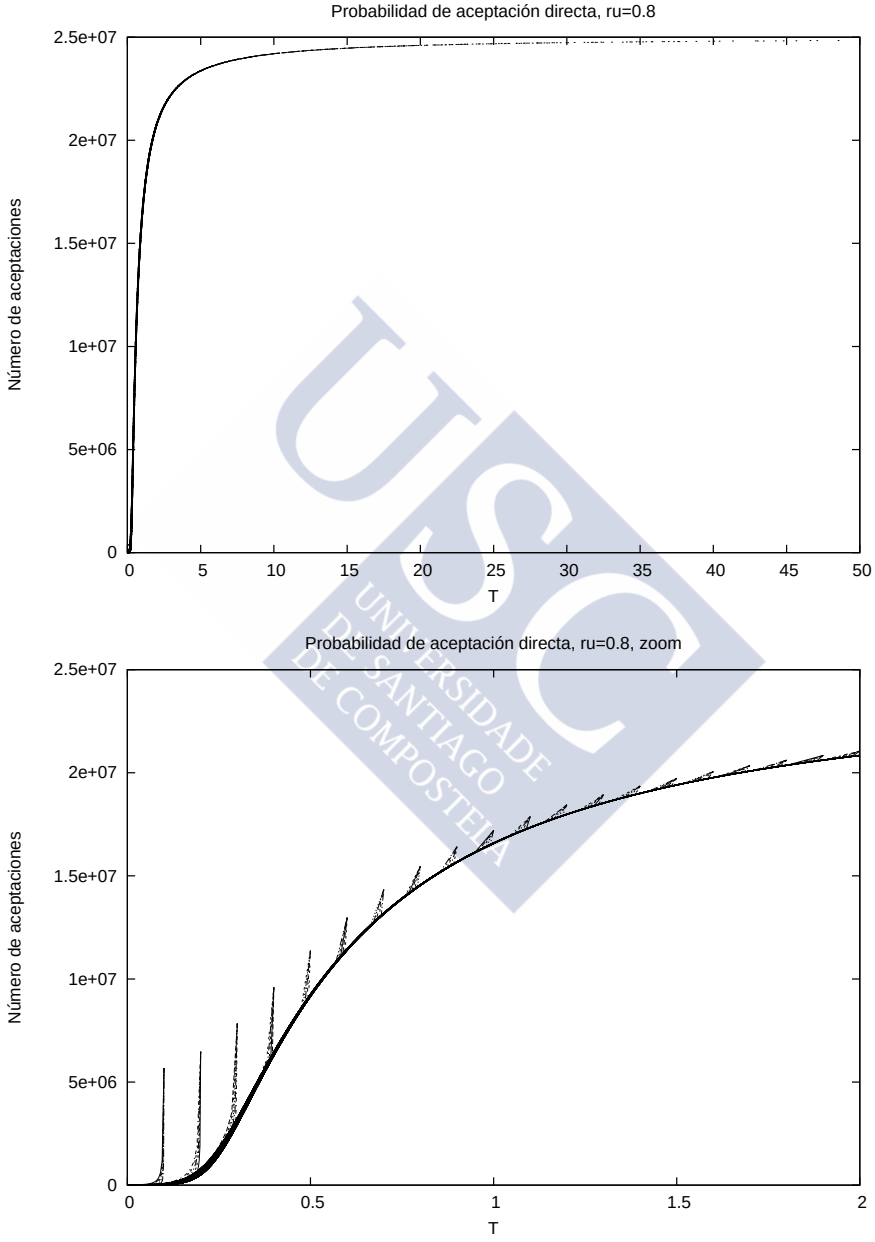

Figura 5.29: Aceptación directa para la familia de curvas con ru=0.8.

partir de una instrumentación manual del código fuente permite obtener los valores de métricas y parámetros de rendimiento durante la ejecución de la aplicación en sistemas reales y, con los datos recogidos, realizar un análisis con el que construir un modelo analítico mediante técnicas estadísticas y de selección de modelos. Con TIA sólo es necesario indicar qué forma tienen los términos, a priori, que deberían ser incluidos en la expresión del modelo.

Las expresiones que obtenemos para la evolución temporal de la *aceptación directa* son:

Probabilidad de aceptación directa, caso ru = 0.0

Primera etapa:

$$\begin{aligned} p_a(t) =& (a + c \cdot log(t) + e \cdot (log(t))^2 + g \cdot (log(t))^3 + i \cdot (log(t))^4) / \\ & (1 + b \cdot log(t) + d \cdot (log(t))^2 + f \cdot (log(t))^3 + h \cdot (log(t))^4 + \\ & j \cdot (log(t))^5) \end{aligned} \tag{5.30}$$

Segunda etapa:

$$\begin{aligned} p_a(t) =& (a + c \cdot log(t) + e \cdot (log(t))^2 + g \cdot (log(t))^3 + i \cdot (log(t))^4) / \\ & (1 + b \cdot log(t) + d \cdot (log(t))^2 + f \cdot (log(t))^3 + h \cdot (log(t))^4 + \\ & j \cdot (log(t))^5) \end{aligned} \tag{5.31}$$

Tercera etapa:

$$\begin{aligned} p_a(t) =& (a + c \cdot log(t) + e \cdot (log(t))^2 + g \cdot (log(t))^3 + i \cdot (log(t))^4) / \\ & (1 + b \cdot log(t) + d \cdot (log(t))^2 + f \cdot (log(t))^3 + h \cdot (log(t))^4 + \\ & j \cdot (log(t))^5) \end{aligned} \tag{5.32}$$

Otras etapas:

$$\begin{aligned} p_a(t) =& (a + c \cdot t^{2,5} + e \cdot t + g \cdot t^{2,5} + i \cdot t^2) / \\ & (1 + b \cdot t^{2,5} + d \cdot t + f \cdot t^{2,5} + h \cdot t^2 + j \cdot t^{2,5}) \end{aligned} \tag{5.33}$$

Probabilidad de aceptación directa, caso ru = 1.0

Primera etapa:

$$\begin{aligned} p_a(t) =& (a + c \cdot log(t) + e \cdot (log(t))^2 + g \cdot (log(t))^3 + i \cdot (log(t))^4 + \\ & k \cdot (log(t))^5)/(1 + b \cdot log(t) + d \cdot (log(t))^2 + f \cdot (log(t))^3 + \\ & h \cdot (log(t))^4 + j \cdot (log(t))^5) \end{aligned} \tag{5.34}$$

Segunda etapa:

$$\begin{aligned} p_a(t) =& (a + c \cdot log(t) + e \cdot (log(t))^2 + g \cdot (log(t))^3 + i \cdot (log(t))^4)/ \\ & (1 + b \cdot log(t) + d \cdot (log(t))^2 + f \cdot (log(t))^3 + h \cdot (log(t))^4 + \\ & j \cdot (log(t))^5) \end{aligned} \tag{5.35}$$

Tercera etapa:

$$\begin{aligned} p_a(t) =& (a + c \cdot log(t) + e \cdot (log(t))^2 + g \cdot (log(t))^3 + i \cdot (log(t))^4)/ \\ & (1 + b \cdot log(t) + d \cdot (log(t))^2 + f \cdot (log(t))^3 + h \cdot (log(t))^4 + \\ & j \cdot (log(t))^5) \end{aligned} \tag{5.36}$$

Otras etapas:

$$\begin{aligned} p_a(t) =& (a + c \cdot log(t) + e \cdot (log(t))^2 + g \cdot (log(t))^3 + i \cdot (log(t))^4 + \\ & k \cdot (log(t))^5)/(1 + b \cdot log(t) + d \cdot (log(t))^2 + f \cdot (log(t))^3 + \\ & h \cdot (log(t))^4 + j \cdot (log(t))^5) \end{aligned} \tag{5.37}$$

Probabilidad de aceptación directa, caso ru >0.0 y ru <1.0

Primera etapa:

$$\begin{aligned} p_a(t, ru) =& (a + c \cdot log(t) + e \cdot \cdot ru + g \cdot (log(t))^2 + i \cdot ru^2 + \\ & k \cdot ru \cdot log(t))/(1 + b \cdot log(t) + d \cdot ru + f \cdot (log(t))^2 + \\ & h \cdot ru^2 + j \cdot ru \cdot log(t)) \end{aligned} \tag{5.38}$$

Coeficiente	Primera etapa	Segunda	Tercera	Otras
a	1.32852e+07	1.05095e+07	1.02839e+07	5011.900967
b	0.094470868	0.600001780	0.871920501	-1.77513954
c	1.52051e+07	2.67215e+07	3.08113e+07	-127218.560
d	1.139918181	1.602845683	1.730959439	1.330277399
e	2.17987e+07	3.26511e+07	3.80893e+07	2.57174e+06
f	-0.00600001	0.157781366	0.269195024	-0.63691343
g	6.94199e+06	1.56181e+07	1.90283e+07	-9.7714e+06
h	0.108133160	0.166198359	0.179589208	0.351497550
i	2.03198e+06	3.07780e+06	3.56116e+06	9.81990e+06
j	-0.00351806	-0.00576274	-0.00556418	-0.00257816

Tabla 5.2: Coeficientes de ajuste para el modelo de aceptación directa: caso ru=0.0

Segunda etapa:

$$\begin{aligned} p_a(t,ru) =& (a + c \cdot log(t) + e \cdot ru + g \cdot (log(t))^2 + i \cdot ru^2 + \\ & k \cdot ru \cdot log(t))/(1 + b \cdot log(t) + d \cdot ru + f \cdot (log(t))^2 + \\ & h \cdot ru^2 + j \cdot ru \cdot log(t)) \end{aligned} \tag{5.39}$$

Tercera etapa:

$$\begin{aligned} p_a(t,ru) =& (a + c \cdot log(t) + e \cdot ru + g \cdot (log(t))^2 + i \cdot ru^2 + \\ & k \cdot ru \cdot log(t))/(1 + b \cdot log(t) + d \cdot ru + f \cdot (log(t))^2 + \\ & h \cdot ru^2 + j \cdot ru \cdot log(t)) \end{aligned} \tag{5.40}$$

Otras etapas:

$$p_a(t,ru) = exp(a + b \cdot t + c \cdot t \cdot log(t) + d \cdot t^{2,5} + e \cdot t^{2,5} + f \cdot e^{(}-t)) \tag{5.41}$$

Tenemos una ecuación racional en el mejor de los modelos que hemos obtenido para cada una de las curvas, excepto para el caso general, que obtenemos una exponencial. Los modelos se muestran en las figuras 5.30, 5.31 y 5.32. Cada ecuación tiene entre 6 y 11 coeficientes de ajuste (distintos, aunque se muestren con el mismo nombre en las ecuaciones). Se muestran en las tablas 5.2, 5.3 y 5.4.

Los ajustes proporcionan valores próximos a los datos experimentales, pero presentan ciertas desviaciones.

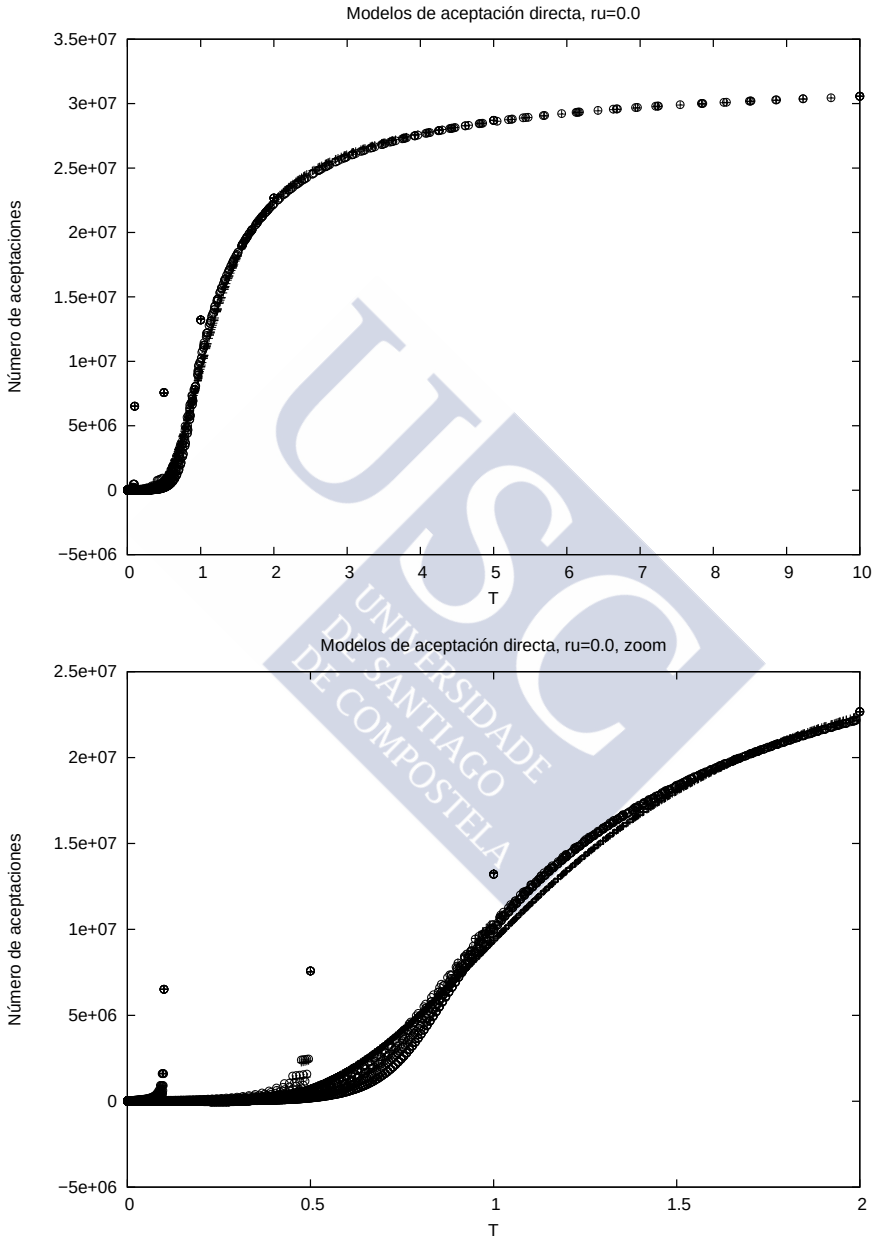

Figura 5.30: Modelo de la probabilidad de aceptación directa para las curvas con ru=0.0.

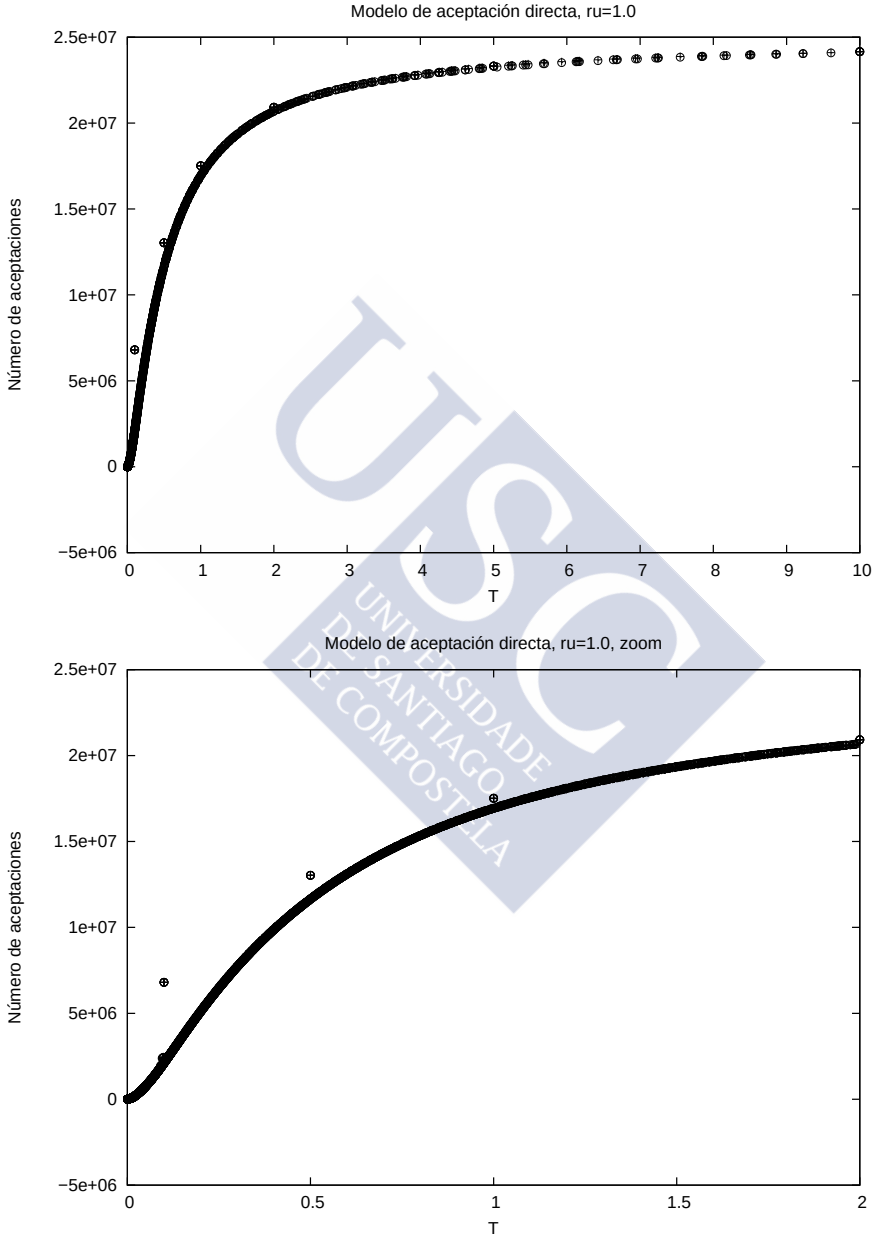

Figura 5.31: Modelo de la probabilidad de aceptación directa para las curvas con ru=1.0.

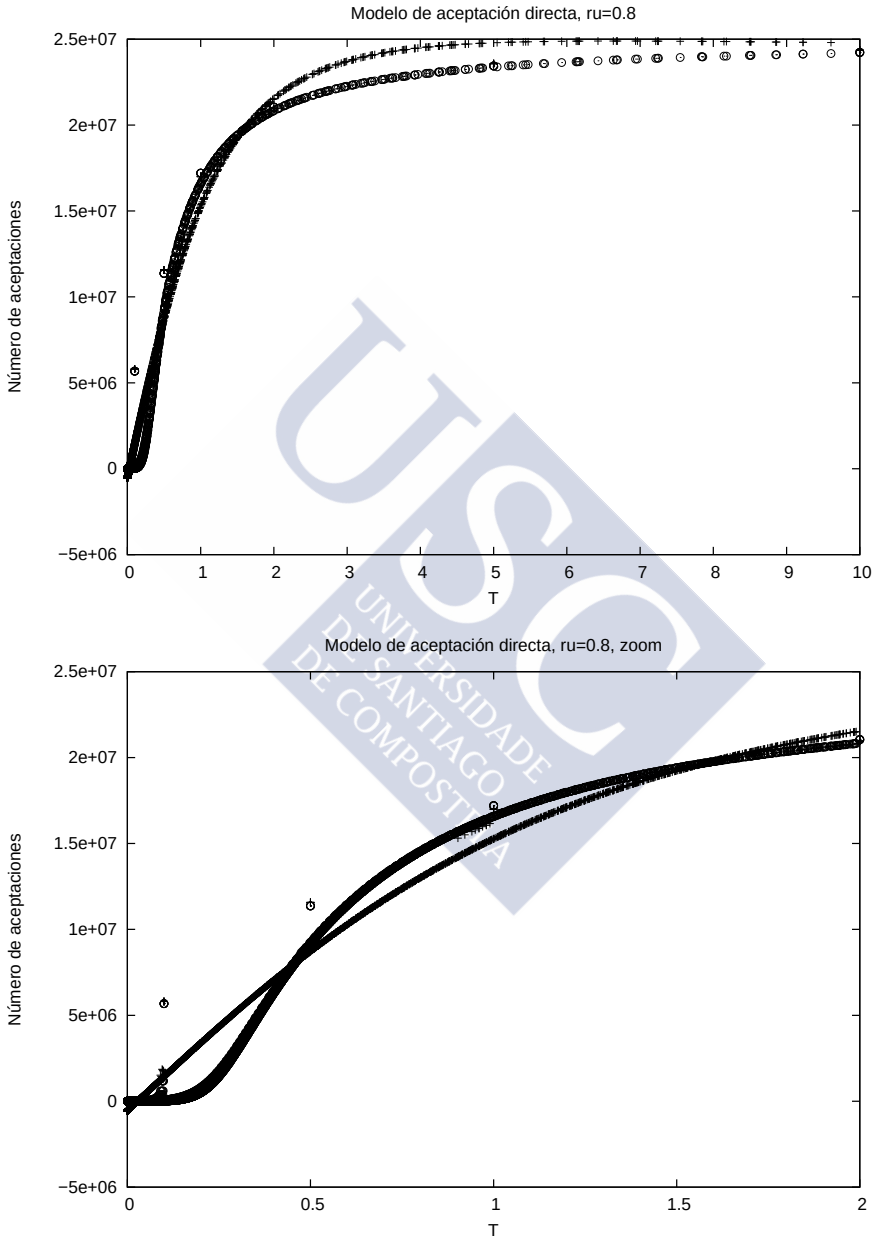

Figura 5.32: Modelo de la probabilidad de aceptación directa para las curvas con ru=0.8.

Coeficiente	Primera etapa	Segunda	Tercera	Otras
a	1.7514e+07	1.69355e+07	1.69343e+07	1.69038e+07
b	-0.36899501	-0.11578746	-0.16266427	0.116729178
c	-498363.033	4.76166e+06	3.96843e+06	8.56618e+06
d	0.091906487	0.128753546	0.136943690	0.151830080
e	-1.8754e+06	-351901.419	-519002.851	1.70931e+06
f	-0.07619532	-0.04392979	-0.04902262	-0.02080636
g	-888092.333	103280.013	-65098.3404	166108.7012
h	0.020887121	0.008428085	0.008709570	0.008404617
i	384711.9254	52988.62423	24709.57387	7851.158393
j	0.003047420	-0.00043142	-0.00054691	-0.00106775
k	84781.50612	-	-	144.2501246

Tabla 5.3: Coeficientes de ajuste para el modelo de aceptación directa: caso ru=1.0

Coeficiente	Primera etapa	Segunda	Tercera	Otras
a	1.04536e+07	8.79244e+06	8.64295e+06	-36.2268671
b	0.176874068	0.267077817	0.281888349	-65.2685501
c	1.0993e+07	1.38722e+07	1.42758e+07	22.74211511
d	-0.19035295	-0.23383279	-0.27885696	-6.07634782
e	5.21621e+06	6.30718e+06	5.78263e+06	118.1998021
f	0.216144323	0.239090834	0.24078797	13.96951685
g	4.54287e+06	5.04576e+06	5.08642e+06	-
h	0.607784996	0.613563503	0.653734355	-
i	9.99998e+06	8.9803e+06	9.59596e+06	-
j	0.31681287	0.345157801	0.34473878	-
k	6.36604e+06	6.67107e+06	6.55414e+06	-

Tabla 5.4: Coeficientes de ajuste para el modelo de aceptación directa: otros casos.

Calculamos también expresiones similares para la evolución de la probabilidad de aceptación por el criterio de Metropolis, con resultados similares. Los modelos se muestran en las figuras 5.33, 5.34 y 5.35, y en las tablas 5.5, 5.6 y 5.7.

Aceptación probabilística, caso ru = 0.0

Primera etapa:

$$\begin{aligned} p_b(t) =& (a + c \cdot log(t) + e \cdot (log(t))^2 + g \cdot (log(t))^3 + i \cdot (log(t))^4) / \\ & (1 + b \cdot log(t) + d \cdot (log(t))^2 + f \cdot (log(t))^3 + h \cdot (log(t))^4 + \\ & j \cdot (log(t))^5) \end{aligned} \tag{5.42}$$

Segunda etapa:

$$\begin{aligned} p_b(t) =& (a + c \cdot log(t) + e \cdot (log(t))^2 + g \cdot (log(t))^3 + i \cdot (log(t))^4) / \\ & (1 + b \cdot log(t) + d \cdot (log(t))^2 + f \cdot (log(t))^3 + h \cdot (log(t))^4 + \\ & j \cdot (log(t))^5) \end{aligned} \tag{5.43}$$

Tercera etapa:

$$\begin{aligned} p_b(t) =& (a + c \cdot log(t) + e \cdot (log(t))^2 + g \cdot (log(t))^3 + i \cdot (log(t))^4) / \\ & (1 + b \cdot log(t) + d \cdot (log(t))^2 + f \cdot (log(t))^3 + h \cdot (log(t))^4 + \\ & j \cdot (log(t))^5) \end{aligned} \tag{5.44}$$

Otras etapas:

$$\begin{aligned} p_b(t) =& (a + c \cdot t^{2,5} + e \cdot t + g \cdot t^{2,5} + i \cdot t^2) / \\ & (1 + b \cdot t^{2,5} + d \cdot t + f \cdot t^{2,5} + h \cdot t^2 + j \cdot t^{2,5}) \end{aligned} \tag{5.45}$$

Aceptación probabilística, caso ru = 1.0

Primera etapa:

$$\begin{aligned} p_b(t) =& (a + c \cdot log(t) + e \cdot (log(t))^2 + g \cdot (log(t))^3) / \\ & (1 + b \cdot log(t) + d \cdot (log(t))^2 + f \cdot (log(t))^3 + h \cdot (log(t))^4) \end{aligned} \tag{5.46}$$

Segunda etapa:

$$\begin{aligned} p_b(t) =& (a + c \cdot log(t) + e \cdot (log(t))^2 + g \cdot (log(t))^3 + i \cdot (log(t))^4) / \\ & (1 + b \cdot log(t) + d \cdot (log(t))^2 + f \cdot (log(t))^3 + h \cdot (log(t))^4 + \\ & j \cdot (log(t))^5) \end{aligned} \tag{5.47}$$

Tercera etapa:

$$p_b(t) = (a + c \cdot log(t) + e \cdot (log(t))^2 + g \cdot (log(t))^3) / (1 + b \cdot log(t) + d \cdot (log(t))^2 + f \cdot (log(t))^3 + h \cdot (log(t))^4) \quad (5.48)$$

Otras etapas:

$$p_b(t) = (a + c \cdot log(t) + e \cdot (log(t))^2 + g \cdot (log(t))^3 + i \cdot (log(t))^4 + k \cdot (log(t))^5) / (1 + b \cdot log(t) + d \cdot (log(t))^2 + f \cdot (log(t))^3 + h \cdot (log(t))^4 + j \cdot (log(t))^5) \quad (5.49)$$

Aceptación probabilística, caso ru >0.0 y ru <1.0

Primera etapa:

$$p_b(t, ru) = (a + c \cdot log(t) + e \cdot ru + g \cdot (log(t))^2 + i \cdot ru^2 + k \cdot ru \cdot log(t)) / (1 + b \cdot log(t) + d \cdot ru + f \cdot (log(t))^2 + h \cdot ru^2 + j \cdot ru \cdot log(t)) \quad (5.50)$$

Segunda etapa:

$$p_b(t, ru) = (a + c \cdot log(t) + e \cdot ru + g \cdot (log(t))^2 + i \cdot ru^2 + k \cdot ru \cdot log(t)) / (1 + b \cdot log(t) + d \cdot ru + f \cdot (log(t))^2 + h \cdot ru^2 + j \cdot ru \cdot log(t)) \quad (5.51)$$

Tercera etapa:

$$p_b(t, ru) = (a + c \cdot log(t) + e \cdot ru + g \cdot (log(t))^2 + i \cdot ru^2 + k \cdot ru \cdot log(t)) / (1 + b \cdot log(t) + d \cdot ru + f \cdot (log(t))^2 + h \cdot ru^2 + j \cdot ru \cdot log(t)) \quad (5.52)$$

Otras etapas:

$$p_b(t, ru) = (a + b \cdot t + c \cdot t^2 + d \cdot t^3 + e \cdot ru) / (1 + f \cdot t + g \cdot t^2 + h \cdot ru) \quad (5.53)$$

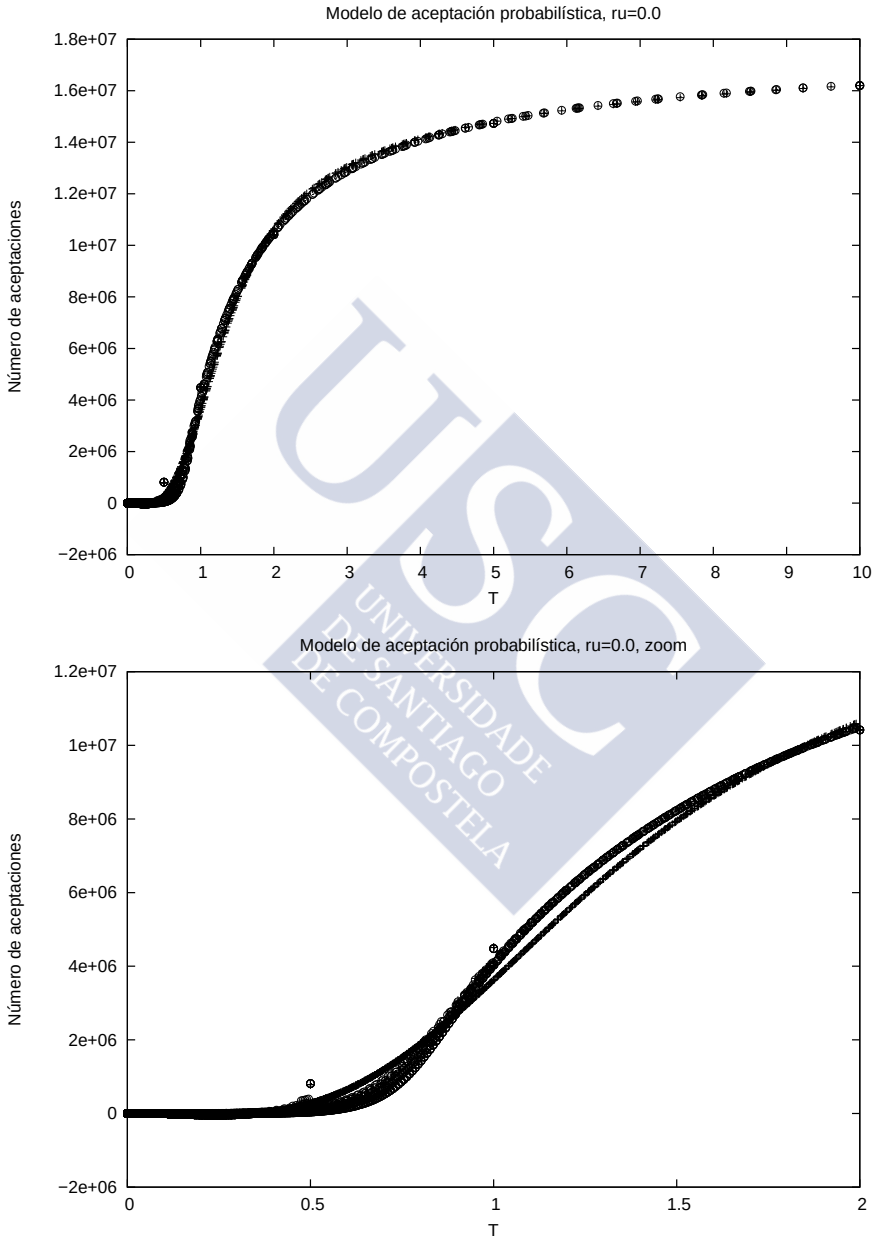

Figura 5.33: Modelo de la probabilidad de aceptación probabilística para las curvas con ru=0.0.

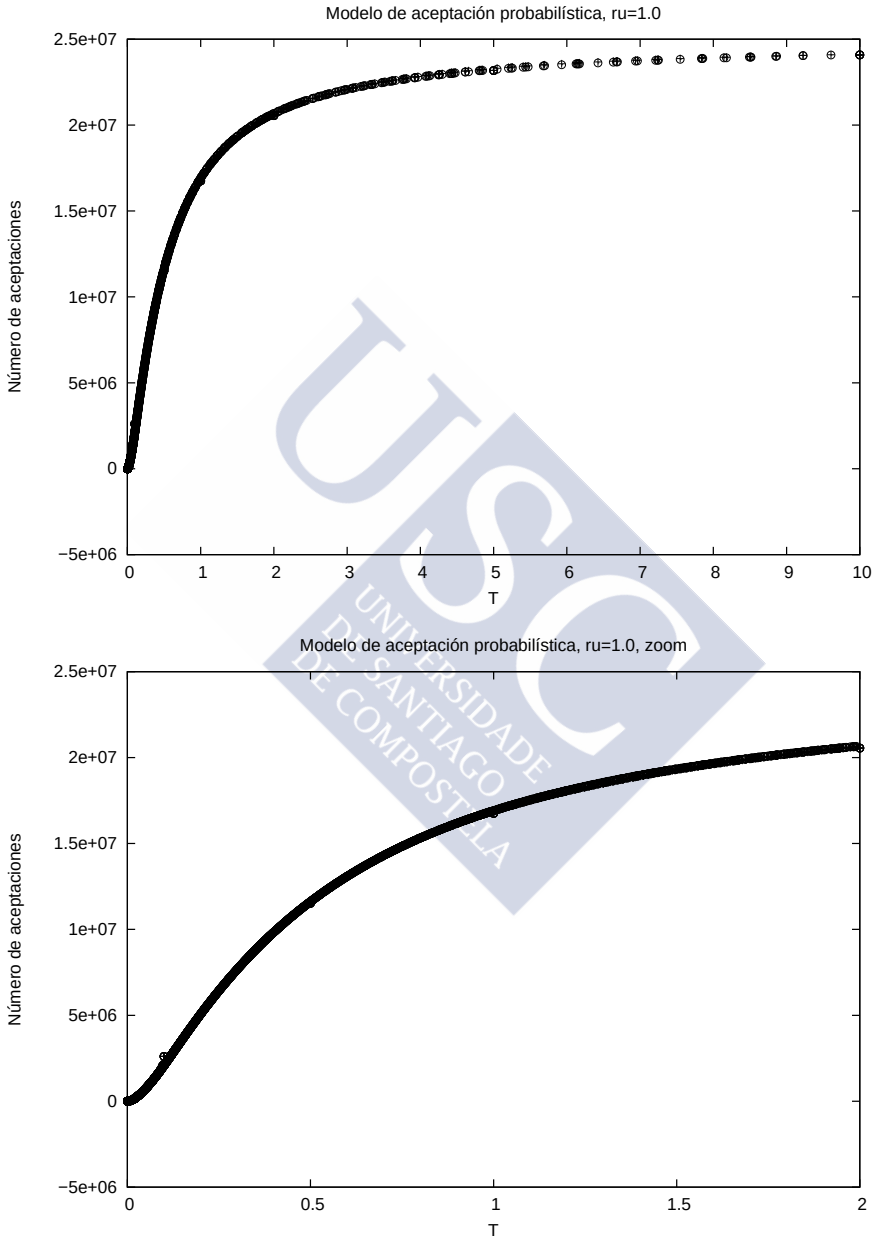

Figura 5.34: Modelo de la probabilidad de aceptación probabilística para las curvas con ru=1.0.

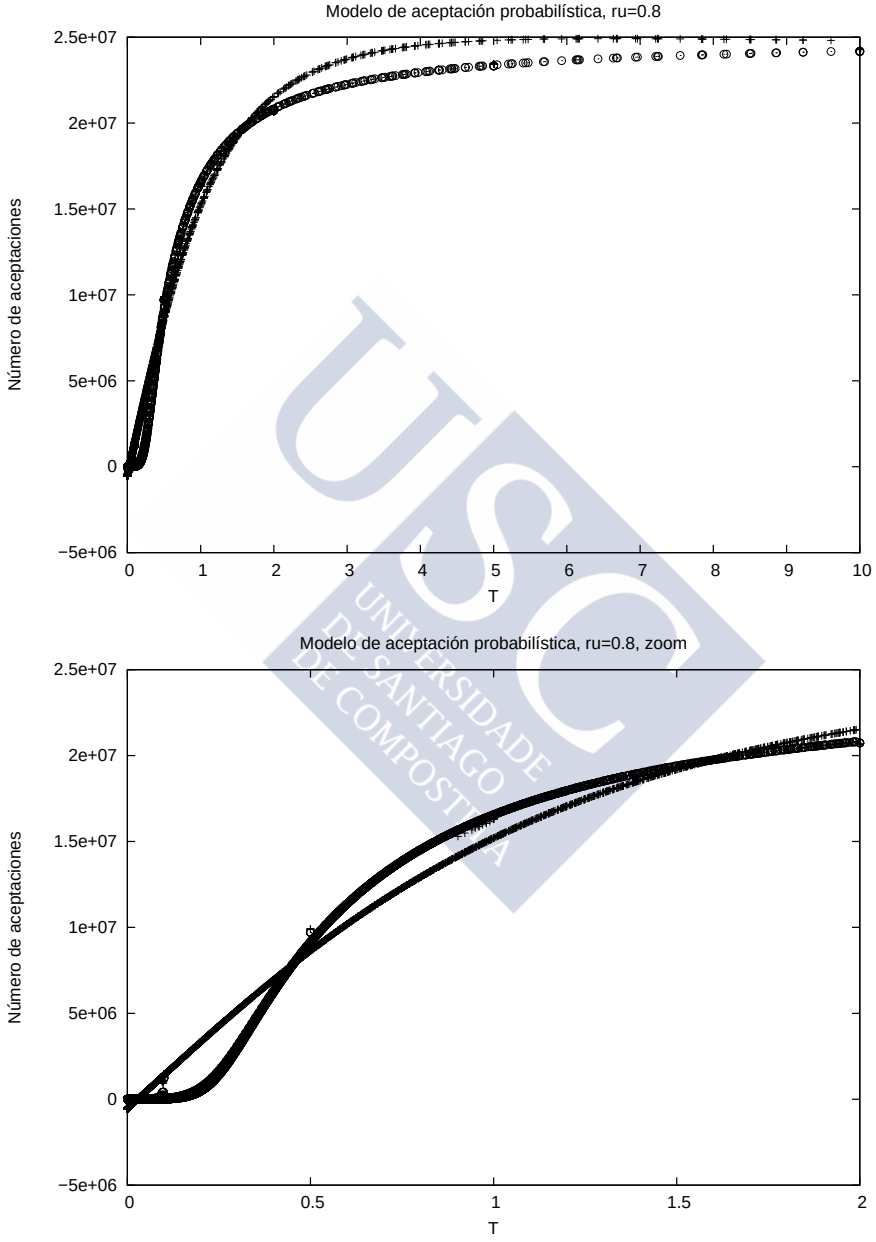

Figura 5.35: Modelo de la probabilidad de aceptación probabilística para las curvas con ru=0.8.

Coeficiente	Primera etapa	Segunda	Tercera	Otras
a	4.50827e+06	4.17125e+06	4.10255e+06	-1559.63816
b	0.033786553	0.557142456	0.703774084	-1.71518838
c	8.73897e+06	1.22725e+07	1.32236e+07	96795.32519
d	1.027871540	1.419656373	1.528320439	1.180719925
e	7.95049e+06	1.44177e+07	1.61013e+07	-196824.981
f	-0.07927952	0.072794782	0.093822835	-0.40245175
g	3.68852e+06	7.36007e+06	8.27501e+06	-1.2498e+06
h	0.100363909	0.151064476	0.170383172	0.171168427
i	657891.9385	1.33168e+06	1.48906e+06	2.19178e+06
j	-0.00523709	-0.00613910	-0.00728756	-0.00251315

Tabla 5.5: Coeficientes de ajuste para el modelo de aceptación probabilística: caso ru=0.0

Coeficiente	Primera etapa	Segunda	Tercera	Otras
a	1.67525e+07	1.69084e+07	1.69105e+07	1.6876e+07
b	0.327506539	-0.02483315	0.304384663	0.118712289
c	1.22393e+07	6.33485e+06	1.18811e+07	8.61395e+06
d	0.193097009	0.229978538	0.179202991	0.151590338
e	3.78231e+06	1.96847e+06	3.32566e+06	1.72019e+06
f	0.011286303	-0.01417797	0.007843883	-0.02052378
g	460921.0201	886889.8347	385975.2707	167154.7873
h	0.000591317	0.015032368	0.000567648	0.008471016
i	-	157034.2434	-	7891.346821
j	-	-0.00063275	-	-0.00109108
k	-	-	-	144.8479355

Tabla 5.6: Coeficientes de ajuste para el modelo de aceptación probabilística: caso ru=1.0

Coeficiente	Primera etapa	Segunda	Tercera	Otras
a	9.37511e+06	8.67565e+06	8.56455e+06	-129538.188
b	0.213342499	0.278731206	0.28739114	5.97133e+06
c	1.24291e+07	1.42054e+07	1.44469e+07	2.9098e+06
d	-0.15563091	-0.24551394	-0.2913607	3856.326195
e	6.30412e+06	6.05229e+06	5.55531e+06	-49569.4164
f	0.215023153	0.239103205	0.241037653	0.136674052
g	4.43861e+06	5.03998e+06	5.08627e+06	0.126132633
h	0.654943785	0.663951293	0.686959517	-0.86179494
i	1.04742e+07	1.00559e+07	1.02663e+07	-
j	0.366808334	0.354280975	0.348332504	-
k	7.70121e+06	6.93103e+06	6.6546e+06	-

Tabla 5.7: Coeficientes de ajuste para el modelo de aceptación probabilística: otros casos.

En resumen, el conjunto total de instrucciones para modelar la evolución temporal del algoritmo de SA secuencial implementado por la herramienta RULES, para las condiciones iniciales fijadas y dentro del rango de parámetros de control tratado, es el siguiente:

Operaciones enteras:

$$\begin{gathered}2\cdot\text{filas}+2\cdot\text{filas}\cdot\text{columnas}+\text{filas}\cdot\text{columnas}\cdot(2+1+p_3\cdot(2+1)+\\ \text{celdas\_utiles}\cdot(3))+2\cdot\text{reordenamientos}+2\cdot\text{reordenamientos}\cdot\text{filas}+\\ 2\cdot\text{reordenamientos}\cdot\text{filas}\cdot\text{columnas}+\text{reordenamientos}\cdot\text{filas}\cdot\text{columnas}\cdot(2+\\ 1+p_3\cdot(2\cdot\text{RANDOMINTI}+2+1+p_3\cdot(2\cdot2)))+\text{filas}\cdot\text{columnas}\cdot(3+\\ 1+p_3\cdot6)+\text{filas}\cdot\text{columnas}\cdot\text{usos}\cdot(6)+2\cdot\text{filas}+2\cdot\text{filas}\cdot\text{columnas}+\\ 2\cdot\text{filas}\cdot\text{columnas}\cdot\text{usos}+4\cdot\text{usos}+2\cdot\text{filas}+2\cdot\text{filas}\cdot\text{columnas}+\\ \text{filas}\cdot\text{columnas}\cdot(3+2\cdot p_3\cdot(1+(1-p_4)\cdot(4+(1-p_8)\cdot2)+p_4\cdot2)+\\ 2\cdot p_3\cdot(1+(1-p_6)\cdot(4+(1-p_8)\cdot2)+p_6\cdot2)+3)+2\cdot\text{usos}+\\ \text{usos}\cdot(6+\text{SQRTI})+\\ \textstyle\sum_t(\text{movimientos\_por\_temperatura}\cdot(1/p_9\cdot(4\cdot\text{RANDOMINTI}+\\ 2\cdot3+3)+2\cdot3+6\cdot4+2\cdot(1+2\cdot((1-p_4)\cdot(3+2\cdot(1+p_8\cdot1)))+\\ 2\cdot((1-p_6)\cdot(3+2\cdot(1+p_8\cdot1)))+4+5+7)+p_a(t,ru)\cdot(2\cdot2)+\\ (1-p_a(t,ru))\cdot(\text{EXPI}+\text{RANDOMI}+p_b(t,ru)\cdot(2\cdot2)+\\ (1-p_a(t,ru)-p_b(t,ru))\cdot(2\cdot2+2+2))))\end{gathered}\tag{5.54}$$

Operaciones en punto flotante:

$$\begin{gathered}\text{reordenamientos}\cdot\text{filas}\cdot\text{columnas}\cdot p_3\cdot2\cdot\text{RANDOMINTF}+3+\\ \text{filas}\cdot\text{columnas}\cdot(p_3\cdot1)+\text{filas}\cdot\text{columnas}\cdot\text{usos}\cdot(1)+\\ 3+\text{usos}\cdot\text{SQRTF}+1+\textstyle\sum_t(\text{movimientos\_por\_temperatura}\cdot(\\ 1/p_9\cdot(4\cdot\text{RANDOMINTF})+6+2+1+(1-p_a(t,ru))\cdot(3+\\ \text{EXPF}+\text{RANDOMF})))\end{gathered}\tag{5.55}$$

donde *RANDOMINTI* y *RANDOMINTF* son las operaciones enteras y en punto flotante de la función *random_int()*; *SQRTI* y *SQRTF* lo son de la función *sqrt()*; *EXPI* y *EXPF*, de la función *exp()*; y *RANDOMI* y *RANDOMF*, de la función *random()*.

5.5. Caracterización de las comunicaciones

El SA ha probado ser una buena técnica para resolver problemas complejos de optimización global. Sin embargo, suele precisar largos tiempos de computación para obtener buenas soluciones. Para mejorar su eficiencia se han probado varias técnicas para paralelizar el modelo básico secuencial en sistemas multiprocesador [187][178].

Presentamos dos estrategias básicas. La primera es utilizar los nodos de computación para acelerar el cálculo de una única simulación del algoritmo, distribuyendo el cálculo entre todos ellos. En este caso, la paralelización busca reducir el impacto computacional de alguna sección de código especialmente crítica del algoritmo secuencial. Posibles ejemplos son funciones de coste que requieran evaluar subobjetivos computacionalmente costosos, o mecanismos de generación de transiciones sometidos a múltiples restricciones que requieren una computación costosa. Esta manera de abordar la paralelización exige un conocimiento detallado del problema y de su implementación, para saber cómo dividirlo y distribuirlo de manera eficiente. Una alternativa consiste en asignar la instancia completa del problema a cada procesador y ejecutar los pasos de la misma optimización secuencial en cada uno (la misma cadena de Markov, para preservar las propiedades de convergencia del algoritmo secuencial). Como, para aplicaciones típicas del algoritmo, un alto porcentaje de las transiciones son rechazadas, y los rechazos son independientes entre sí porque dependen de propiedades globales del estado actual, pueden ejecutarse independientemente en distintos procesadores. En este caso es necesario realizar una sincronización entre los procesadores cuando alguno llegue a aceptar una transición. La eficiencia dependerá del volumen de movimientos aceptados.

La otra estrategia básica de paralelización consiste en utilizar los nodos de computación para ejecutar simultáneamente distintas simulaciones del algoritmo secuencial para el mismo problema con la misma o diferentes combinaciones de parámetros de entrada y control. Puesto que el proceso es estocástico, diferentes ejecuciones producirán diferentes soluciones cerca del óptimo entre las que posteriormente se puede seleccionar la mejor. La ventaja de esta implementación paralela frente a la secuencial es que, simultáneamente, se hacen tantas simulaciones como nodos haya disponibles. Al ser un proceso de optimización estocástico las posibilidades de encontrar una mejor solución con respecto al caso secuencial se multiplican por el número de nodos, en el mismo tiempo.

La implementación secuencial del SA que se examina en este capítulo no realiza cálculos costosos en cada iteración, por lo que la mejor manera de aprovechar un sistema de cálculo distribuido es esta última estrategia de paralelismo de grano grueso. Se han diseñado ocho diferentes posibilidades de utilizar el SA secuencial en el sistema paralelo. La contabilidad de operaciones enteras y de punto flotante reaprovecha los resultados obtenidos para el caso secuencial, aunque se ignora una pequeña sobrecarga debida a la contribución de las operaciones asociadas a las etapas de selección del mejor resultado entre todos los nodos y a la transmisión al nodo raíz del resultado aceptado. Se ha añadido la evaluación de las comunicaciones entre nodos para las distintas opciones.

Opción A

Esta primera implementación ejecuta el mismo código secuencial en todos los elementos de computación con la misma configuración de parámetros. Sin embargo las evoluciones temporales del proceso no son idénticas. Debido a la naturaleza estocástica del SA, diferentes ejecuciones proporcionan soluciones ligeramente diferentes cerca del óptimo. Disponer de una colección de buenos valores permite escoger el mejor. Como la condición de terminación de este algoritmo hace referencia al número máximo de movimientos posibles y no al valor del coste, el proceso termina de forma prematura y no está garantizado obtener el mejor resultado posible. La ventaja principal de este código paralelo es obtener un mejor resultado que en una única ejecución secuencial.

El resultado de esta implementación paralela es equivalente a ejecutar el código original consecutivamente tantas veces como nodos haya disponibles, comparar los resultados y seleccionar el mejor. Pero con un factor de reducción en el tiempo de ejecución aproximadamente igual a *1/nodos*. Y, con una probabilidad *(nodos-1)/nodos* de obtener un resultado mejor que en una única ejecución secuencial.

El proceso completo es el siguiente:

– Todos los nodos cargan simultáneamente los mismos datos de entrada para resolver el mismo problema; alternativamente, el nodo raíz carga el problema y lo distribuye al resto mediante una operación *broadcast* (para un sistema con paralelismo por pase de mensajes)

– El proceso secuencial se ejecuta en todos los nodos simultáneamente hasta alcanzar un resultado

– Todos los nodos envían a todos los demás el valor del coste del estado final (no el estado) mediante una operación *allgather*; alternativamente, puede utilizar *gather* seguido de *broadcast*

– Ahora todos los nodos conocen el coste de las soluciones en todos los demás. El nodo con el mejor resultado de coste envía su estado final al nodo raíz con una operación *send*, mientras que el resto de nodos no hace nada

Alternativamente, en lugar de utilizar una operación *allgather*, los nodos pueden enviar sus resultados parciales al nodo raíz utilizando la operación *gather*, el nodo raíz selecciona el mejor resultado entre ellos, avisa a todos los nodos quién contiene el mejor resultado mediante *broadcast*, y el seleccionado envía al raíz el estado mediante *send*.

Todos los nodos ejecutan el algoritmo secuencial básico. El número de operaciones enteras y en punto flotante a partir de las ecuaciones calculadas anteriormente, utilizando como valores de entrada la combinación particular replicada en todos los nodos.

Las comunicaciones necesarias en esta implementación son debidas a: la publicación inicial de los parámetros de entrada y control desde el nodo raíz a todos los demás (múltiples valores), la comunicación entre todos los nodos de los valores de coste (valor en punto flotante) y el envío al nodo raíz del estado desde el mejor nodo (valores enteros). Se ignora la primera comunicación suponiendo que los parámetros de entrada se proporcionan a cada proceso en tiempo de inicialización o que se leen localmente en cada nodo, y se modelan sólo las comunicaciones a partir de que comienza el proceso iterativo.

En total, para un sistema de N nodos, utilizando *gather* más *broadcast*, se obtienen las siguientes comunicaciones:

– *gather*: N de un número en punto flotante

– *broadcast*: N de un número entero

– *send*: 1 mensaje de tamaño $filas \cdot columnas$ de números enteros

Opción B

En esta implementación el mismo código secuencial es ejecutado por todos los nodos, con los mismos valores para los parámetros de entrada *ct* (constante de enfriamiento) y *ru* (coeficientes de la función de coste), y diferentes valores para el parámetro *tz* (temperatura inicial).

Ejecutar los códigos con los mismos parámetros iniciales genera procesos de optimización diferentes debido a la aleatoriedad propia del algoritmo. Pero ejecutar con parámetros iniciales diferentes, permite disponer de una exploración más extensa de estados en las etapas iniciales, facilitando que la convergencia a la solución de cada uno de los procesos utilice rutas (cadenas de estados) radicalmente diferentes a las utilizadas en el primer caso.

El resultado es equivalente a utilizar el código original consecutivamente tantas veces como nodos haya, con configuraciones de parámetros ligeramente diferentes. Puesto que el criterio de parada está señalado por un límite superior en el número de movimientos y el coste computacional de aceptaciones y rechazos es aproximadamente el mismo, la versión paralela obtiene un factor de reducción en el tiempo de ejecución igual a *1/nodos*, y una probabilidad *(nodos-1)/nodos* de obtener un resultado mejor que en la única ejecución secuencial.

El proceso completo es similar al del caso anterior:

- Suponemos que cada nodo conoce los valores de sus parámetros de entrada
- El proceso secuencial se ejecuta en todos los nodos simultáneamente hasta alcanzar un resultado
- Todos los nodos envían al raíz el valor del coste del estado final (no el estado) mediante una operación *gather*
- El nodo raíz selecciona el mejor resultado y lo comunica mediante una operación *broadcast*
- El nodo con el mejor resultado de coste envía su estado final al nodo raíz con una operación *send*

Se puede calcular el número de operaciones enteras y en punto flotante a partir de las ecuaciones calculadas anteriormente, utilizando para cada nodo los valores correspondientes, y

ofrecer un valor máximo y un valor mínimo. Nótese que, como el modelo obtenido no depende del valor del parámetro de control *tz*, las ecuaciones proporcionan los mismos resultados para todos los nodos.

Las comunicaciones que aparecen en esta implementación son las mismas del caso anterior.

Opción C

En esta implementación el mismo código secuencial es ejecutado por todos los nodos, con los mismos valores para los parámetros de entrada *tz* y *ru*, y diferentes valores para el parámetro *ct*.

El análisis es similar al de la opción B.

La ventaja básica de este algoritmo paralelo frente a la versión secuencial es un mayor número de resultados de los que escoger el mejor, lo que aumenta la probabilidad de localizar el óptimo. La ventaja frente a la primera versión paralela es una mayor exploración inicial, lo que genera una convergencia hacia la solución desde estados mucho más dispares entre sí. Frente a la segunda versión paralela, realiza exploraciones en las etapas iniciales con diferentes valores de constante de enfriamiento, lo que da lugar a distintas velocidades de disminución de la temperatura y distintas aceptaciones.

El número de operaciones enteras y en punto flotante calculadas para esta implementación puede calcularse a partir de las ecuaciones del modelo, que resultan el mismo para todos los nodos porque no dependen del valor inicial del parámetro *ct*.

El número de comunicaciones es similar al caso anterior.

Opción D

En esta implementación el mismo código secuencial es ejecutado por todos los nodos, con los mismos valores para los parámetros de entrada *tz* y *ct*, y diferentes valores para el parámetro *ru*. El análisis es similar a los dos anteriores.

Esta implementación no tiene ventajas evidentes frente a las dos siguientes versiones paralelas, ya que realiza la misma exploración en las etapas iniciales, pero a partir de variaciones en un parámetro diferente.

El número de operaciones enteras y en punto flotante calculadas para esta implementación puede calcularse a partir de las ecuaciones del modelo. Las probabilidades de aceptación y rechazo de movimientos dependen del valor del parámetro *ru*, por lo que en cada nodo se obtendrán números diferentes de operaciones y esta implementación ofrece un valor máximo y un valor mínimo para ellas.

El número de comunicaciones es similar a los casos anteriores.

Opción E

En esta propuesta el mismo código secuencial se ejecuta en todos los nodos, con los mismos valores para los parámetros de entrada *ct* y *ru*, y diferentes valores para el parámetro *tz*, igual que en la opción B. Sin embargo, al alcanzar un número prefijado de temperaturas los resultados de coste en cada nodo se comparan, se selecciona el mejor nodo y se replica su estado al resto. El proceso continúa y al terminar de nuevo se selecciona el mejor resultado entre todos los nodos.

Esta opción combina las fortalezas de las dos primeras propuestas. Por un lado, permite una exploración extensa en las etapas iniciales usando simultáneamente diferentes procesos de optimización con diferentes parámetros de entrada, que examinan en conjunto más estados que el código secuencial básico. Por otro, cuando el balanceo exploración-convergencia empieza a inclinarse por la segunda, todos los nodos reciben el mismo estado y siguen procesándolo, lo que supera al código secuencial básico en probabilidad de localizar la mejor solución. La ventaja principal de este método es que localiza y repite una buena configuración para el proceso en todos los nodos. Adicionalmente, el usuario no necesita proporcionar la mejor combinación de parámetros de entrada para localizarla sino que el procedimiento paralelo busca e identifica la mejor y trabaja a partir de ella.

La elección del número prefijado de temperaturas es un compromiso entre la oportunidad de ampliar la búsqueda en los estados iniciales y la necesidad de proporcionar suficientes etapas posteriores.

El resultado de esta implementación en paralelo equivale a ejecutar el código original consecutivamente tantas veces como nodos haya disponibles con distintas configuraciones de entrada, detenerlos en un momento predeterminado, comparar los resultados y quedarse con el mejor. Y a continuación volver a ejecutarlo el mismo número de veces, usando como pa-

rámetros de entrada los que antes han producido el mejor resultado y como estado inicial ese mismo resultado.

El proceso completo es:

- El proceso secuencial se ejecuta en todos los nodos simultáneamente durante m temperaturas
- Todos los nodos envían al raíz el valor del coste del estado final (no el estado) mediante una operación *gather*
- El nodo raíz selecciona el mejor resultado y lo comunica mediante una operación *broadcast*
- El nodo con el mejor resultado de coste envía a todos los nodos su estado actual, así como cualquier otra información necesaria para permitir continuar la ejecución en ese punto, mediante una operación *broadcast*
- El proceso secuencial continúa en todos los nodos simultáneamente hasta finalizar
- Todos los nodos envían al raíz el valor del coste del estado final mediante una operación *gather*
- El nodo raíz selecciona el mejor resultado y lo comunica mediante una operación *broadcast*
- El nodo con el mejor resultado de coste envía su estado final al nodo raíz con una operación *send*

Se puede calcular el número de operaciones enteras y en punto flotante a partir de las ecuaciones calculadas anteriormente, utilizando para cada nodo los valores correspondientes, y ofrecer un valor máximo y un valor mínimo. Sin embargo sólo un proceso realiza una evolución básica, mientras que $N-1$ procesos cambian de proceso de optimización después de un número prefijado de temperaturas.

Nótese que la evolución de estos procesos está dividida en etapas diferenciadas: la primera, donde evolucionan desde distintos configuraciones iniciales de 0 a m temperaturas, cada nodo

con operaciones diferentes; y la segunda, desde la temperatura m hasta la temperatura final M, donde todos los nodos realizan las mismas operaciones.

Sin pérdida de generalidad, si se supone que es el nodo 0 el que obtiene el mejor resultado al alcanzar el número prefijado de temperaturas, su evolución corresponde directamente al caso secuencial y se puede calcular el número de operaciones enteras y en punto flotante aplicando directamente las ecuaciones del modelo:

$$\begin{gathered}\text{operaciones}_0(\text{temperatura } M) = \\ \text{modelo}(\text{configuración inicial nodo } 0, \text{temperaturas } 0 \text{ a } M)\end{gathered} \tag{5.56}$$

Se puede dividir en dos partes diferenciadas:

$$\begin{gathered}\text{operaciones}_0(\text{temperatura } M) = \\ \text{modelo}(\text{configuración inicial nodo } 0, \text{temperaturas } 0 \text{ a } m) + \\ \text{modelo}(\text{configuración inicial nodo } 0, \text{temperaturas } m \text{ a } M)\end{gathered} \tag{5.57}$$

Las ecuaciones obtenidas en el apartado anterior sólo proporcionan valores acumulados, así que se obtiene el número de operaciones entre las temperaturas m y M de los valores de 0 a m y de 0 a M:

$$\begin{gathered}\text{modelo}(\text{configuración inicial nodo } 0, \text{temperaturas } m \text{ a } M) = \\ \text{modelo}(\text{configuración inicial nodo } 0, \text{temperaturas } 0 \text{ a } M) - \\ \text{modelo}(\text{configuración inicial nodo } 0, \text{temperaturas } 0 \text{ a } m)\end{gathered} \tag{5.58}$$

Para el resto de nodos la primera parte es particular y única, mientras que la segunda coincide con la evolución del nodo 0. Las operaciones resultantes son:

$$\begin{gathered}\text{operaciones}_i(\text{temperatura } M) = \\ \text{modelo}(\text{configuración inicial nodo } i, \text{temperaturas } 0 \text{ a } m) + \\ \text{modelo}(\text{configuración inicial nodo } 0, \text{temperaturas } m \text{ a } M)\end{gathered}$$

$$\begin{gathered}\text{operaciones}_i(\text{temperatura } M) = \\ \text{modelo}(\text{configuración inicial nodo } i, \text{temperaturas } 0 \text{ a } m) + \\ \text{modelo}(\text{configuración inicial nodo } 0, \text{temperaturas } 0 \text{ a } M) - \\ \text{modelo}(\text{configuración inicial nodo } 0, \text{temperaturas } 0 \text{ a } m)\end{gathered} \tag{5.59}$$

Sin embargo, no se conoce a priori cuál va a ser el nodo que obtendrá el mejor resultado al alcanzar el número prefijado de temperaturas y que se replicará en el resto. Por tanto, de la misma manera que se ha calculado para el nodo 0, se debe realizar el cálculo para todos los nodos y así se puede proporcionar unos valores máximos y mínimos para el número de operaciones enteras y en punto flotantes. En este caso particular donde el parámetro de entrada que sirve para diferenciar los distintos procesos es la temperatura inicial, este cálculo se simplifica ya que las ecuaciones no dependen de ese parámetro. Por tanto, los valores máximos y mínimos coinciden entre sí y coinciden con el valor del algoritmo secuencial básico.

Las comunicaciones necesarias en el proceso completo son: la comunicación de todos los nodos al raíz de los valores de coste (valor en punto flotante), la comunicación del mejor (valor entero), el envío del estado desde el mejor nodo a todos los demás (valores enteros y en punto flotante), la comunicación de todos los nodos al raíz de los valores de coste (valor en punto flotante), la comunicación del mejor (valor entero), y el envío al nodo raíz del estado desde el mejor nodo (valores enteros).

En total, para un sistema de N nodos, se obtienen:

- N mensajes de un número en punto flotante
- N mensajes de un número entero
- N mensajes de tamaño *filas*$\cdot$ *columnas* de números enteros (estado), $3 \cdot N$ mensajes de un número en punto flotante (temperatura, coste y resultado parcial de aptitud) y N mensajes de un número entero (resultado parcial de compacidad)
- N mensajes de un número en punto flotante
- N mensajes de un número entero
- 1 mensaje de tamaño *filas*$\cdot$ *columnas* de números enteros

Opción F

Esta implementación es idéntica a la opción E, pero escogiendo como parámetro variable ct.

De forma similar al caso anterior, la ventaja básica de este algoritmo paralelo frente a la versión secuencial es una exploración inicial más amplia buscando buenas condiciones de partida seguida de una convergencia más rápida. Esa búsqueda en las etapas iniciales es también la mayor ventaja frente a las primeras alternativas paralelas. No tiene ventajas evidentes frente al caso anterior, sino que realiza la misma exploración en las etapas iniciales, pero a partir de variaciones en otro de los parámetros.

El número de operaciones enteras y en punto flotante coinciden con el caso secuencial, ya que los modelos no dependen del parámetro *ct* inicial.

El número de comunicaciones es similar a la opción anterior.

Opción G

En este caso, el parámetro que varía entre los diferentes nodos es *ru*. Esta opción no tiene ventajas evidentes frente a los dos casos anteriores, sino que realiza la misma exploración en las etapas iniciales, pero a partir de variaciones en otro de los parámetros.

El número de operaciones enteras y en punto flotante ya no coinciden con el caso secuencial. Cuando se selecciona el mejor resultado parcial después del número prefijado de temperaturas, se copia el estado en todos los nodos. Y como todos los nodos tienen un valor diferente del parámetro *ru*, las ecuaciones del modelo proporcionan un número de operaciones diferente para cada uno. Se puede calcular el número de operaciones en cada nodo hasta ese instante, como:

$$\begin{gathered}\text{operaciones}_i(\text{temperatura } m) = \\ \text{modelo}(\text{configuración inicial nodo } i, \text{temperaturas 0 a } m)\end{gathered} \tag{5.60}$$

La segunda fase, desde la temperatura m a la temperatura M, depende de cuál sea el nodo que se replique en todos los demás, que no se conoce a priori. Se pueden establecer valores máximo y mínimos:

$$\begin{gathered}\text{modelo}(\text{configuración inicial nodo } i, \text{temperaturas 0 a } m) + \\ \text{máx}(\text{modelo}(\text{configuración inicial todos los nodos}, \text{temperaturas } m \text{ a } M)) \\ \leq \text{operaciones}_i(\text{temperatura } m) \leq \\ \text{modelo}(\text{configuración inicial nodo } i, \text{temperaturas 0 a } m) + \\ \text{máx}(\text{modelo}(\text{configuración inicial todos los nodos}, \text{temperaturas } m \text{ a } M))\end{gathered} \tag{5.61}$$

El número de comunicaciones es similar a los dos casos anteriores.

Opción H

En esta propuesta, el mismo código secuencial se ejecuta en todos los nodos con la misma configuración de parámetros. Periódicamente, cada m temperaturas, los nodos identifican cuál contiene el mejor resultado de coste, y el estado correspondiente se copia a todos ellos.

Esta implementación está orientada a mantener el procedimiento de convergencia convenientemente enfocado, purgando periódicamente de los nodos los procesos de optimización que no sean óptimos a costa de un incremento en el número de comunicaciones. Respecto a las alternativas anteriores este método pierde la primera fase de exploración ampliada (explorar desde diferentes combinaciones de parámetros de entrada) y la sustituye por la del algoritmo básico (explorar desde una única combinación). Sin embargo, como se ejecuta en cada nodo de manera independiente, la exploración inicial sigue siendo superior al caso secuencial.

El proceso completo es:

- El proceso secuencial se ejecuta en todos los nodos simultáneamente durante m temperaturas
- Todos los nodos envían al raíz el valor del coste del estado final (no el estado) mediante una operación *gather*
- El nodo raíz selecciona el mejor resultado y lo comunica mediante una operación *broadcast*
- El nodo con el mejor resultado de coste envía a todos los nodos su estado actual, así como cualquier otra información necesaria para permitir continuar la ejecución en ese punto, mediante una operación *broadcast*
- El proceso secuencial continúa en todos los nodos simultáneamente durante *m* temperaturas
- Se repite el proceso iterativamente hasta el final
- Todos los nodos envían al raíz el valor del coste del estado final (no el estado) mediante una operación *gather*

- El nodo raíz selecciona el mejor resultado y lo comunica mediante una operación *broadcast*
- El nodo con el mejor resultado de coste envía a todos los nodos su estado actual, así como cualquier otra información necesaria para permitir continuar la ejecución en ese punto, mediante una operación *broadcast*
- El proceso secuencial continúa en todos los nodos simultáneamente durante m temperaturas
- ...
- Todos los nodos envían al raíz el valor del coste del estado final mediante una operación *gather*
- El nodo raíz selecciona el mejor resultado y lo comunica mediante una operación *broadcast*
- El nodo con el mejor resultado de coste envía su estado final al nodo raíz con una operación *send*

La cantidad de veces que se repite este proceso depende de la relación entre el número prefijado de temperaturas que fuerzan una sincronización de los nodos, m, y el número máximo de temperaturas que aparece en el criterio de parada del algoritmo secuencial, M. En total se repite $\lceil m/M \rceil$ veces.

El número de operaciones enteras y en punto flotante de manera general no coinciden con el caso secuencial. Cada vez que se selecciona el mejor resultado parcial, después del número prefijado de temperaturas, se copia el estado en todos los nodos. Si todos los nodos tienen un valor diferente de los parámetros de entrada, las ecuaciones del modelo proporcionan un número de operaciones diferente para cada uno. Y se debe calcular el número de operaciones generalizando el procedimiento del caso anterior: calculando las operaciones en los nodos por bloques de temperaturas, y proporcionando números máximos y mínimos. Sin embargo, en este caso los nodos utilizan los mismos valores en los parámetros de entrada, lo que permite calcular el número de operaciones enteras y en punto flotante aplicando las ecuaciones del modelo directamente.

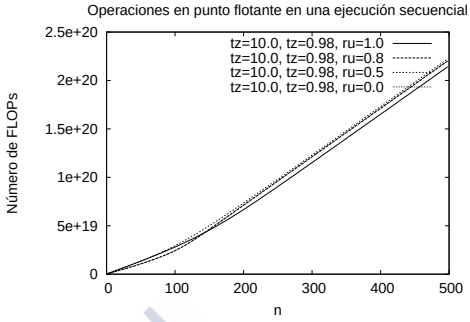

Figura 5.36: Número de FLOPs en una ejecución secuencial. No todas estrategias de paralelización generan el mismo número de operaciones en punto flotante (depende del valor concreto de los parámetros de control), pero son del mismo orden de magnitud (en ningún procesador se calcula más de un proceso de optimización) y comparten el comportamiento respecto al número de procesadores.

El número de comunicaciones es: la comunicación de todos los nodos al raíz de los valores de coste (valor en punto flotante), la comunicación del mejor (valor entero), el envío del estado desde el mejor nodo a todos los demás (valores enteros y en punto flotante), todo ello repetido $\lceil m/M \rceil$ veces; y por último, la comunicación de todos los nodos al raíz de los valores de coste (valor en punto flotante), la comunicación del mejor (valor entero), y el envío al nodo raíz del estado desde el mejor nodo (valores enteros).

En total, para un sistema de N nodos, se obtienen:

- (N mensajes de tamaño un número en punto flotante, N mensajes de un número entero, N mensajes de tamaño *filas*$\cdot$ *columnas* de números enteros (estado), $3 \cdot N$ mensajes de un número en punto flotante (temperatura, coste y resultado parcial de aptitud) y N mensajes de un número entero (resultado parcial de compacidad)), multiplicado por $\lceil m/M \rceil$.
- N mensajes de un número en punto flotante
- N mensajes de un número entero
- 1 mensaje de tamaño *filas*$\cdot$ *columnas* de números enteros

En la figura 5.36 se muestra el comportamiento temporal de los FLOPs en el algoritmo secuencial para un conjunto de parámetros de control. En la segunda gráfica se muestra, para

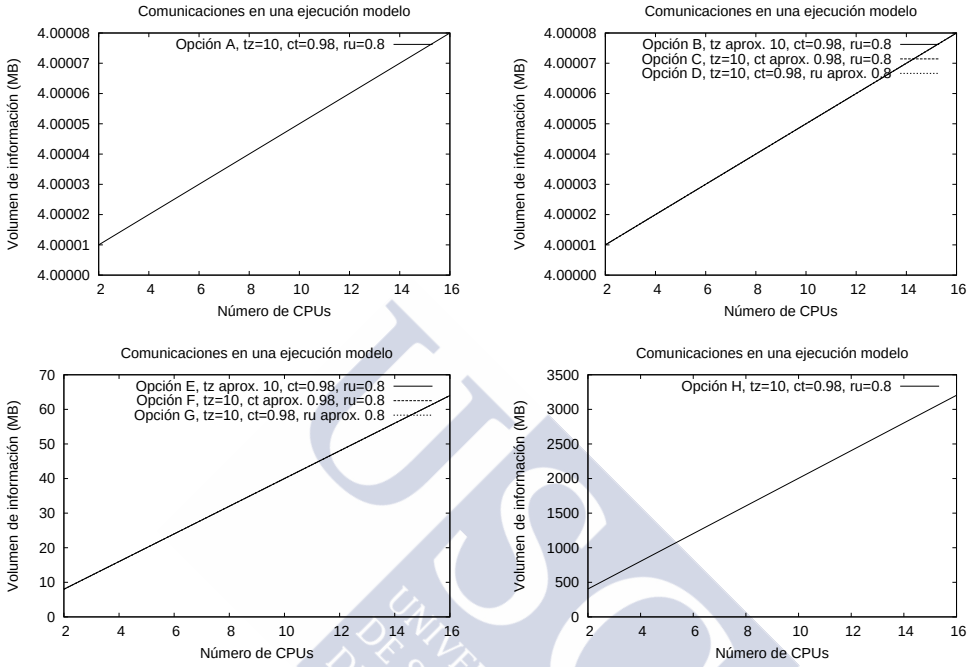

Figura 5.37: Volumen de comunicaciones en la ejecución paralela. Se muestran el número de MB que viajan por la red en las opciones A a H.

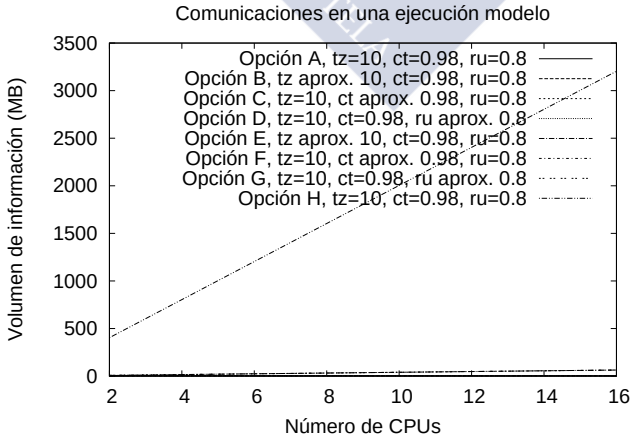

Figura 5.38: Comparación del volumen de comunicaciones en la ejecución paralela en las opciones A a H.

una de las implementaciones paralelas del algoritmo, el número total de FLOPs en función del número de procesadores del sistema. En la figura 5.37 se muestra el volumen de comunicaciones para las distintas implementaciones paralelas del algoritmo, que se superponen en la figura 5.38 para facilitar su comparación.

Conclusiones

La evolución de las ciencias de la computación permite abordar problemas en ciencia e ingeniería cada vez mayores, más complejos, con mejor resolución, y obtener soluciones en un tiempo razonable. Las arquitecturas distribuidas juegan un papel protagonista en este escenario. Sin embargo, la complejidad intrínseca de estos sistemas dificulta el desarrollo de aplicaciones que aprovechen eficientemente los recursos computacionales. Se observa una necesidad de nuevas herramientas y metodologías de análisis, que permitan comprender y anticipar el comportamiento del código en las arquitecturas paralelas de manera más intuitiva.

El método de predicción de rendimiento es una técnica que se utiliza en estas arquitecturas para ayudar a planificar la distribución de trabajos a lo largo de todo el sistema computacional. Pero hay una categoría de problemas, llamados códigos irregulares, cuyo comportamiento depende fuertemente de los datos de entrada y no pueden ser caracterizados correctamente de forma estática. Estos problemas aparecen con frecuencia en el software científico y de ingeniería.

En este trabajo hemos abordado un grupo de algoritmos que representan distintas categorías de códigos irregulares, con una metodología basada en mediciones exhaustivas de las ejecuciones en entornos controlados con una gran variedad los parámetros de entrada, para obtener expresiones analíticas simples del rendimiento que permitan predecir con precisión diferentes escenarios. El objetivo es correlacionar resultados de rendimiento y los valores de las magnitudes monitorizadas. El trabajo ha supuesto distintas aproximaciones al problema de la modelización de códigos irregulares tanto en el aspectos de las computaciones como en el de las comunicaciones.

La aportación principal de este trabajo es el uso de una metodología basada en mediciones intensivas para extraer los parámetros relevantes de cada aplicación, así como la aproximación particular a cada código irregular para llegar a los modelos de rendimiento finales en términos de computaciones y comunicaciones.

Los códigos estudiados incluyen una librería de comunicaciones MPI orientada al Grid, una librería matemática para la resolución de sistemas lineales mediante métodos iterativos, un simulador de la dispersión atmosférica de contaminantes, y un algoritmo de optimización estocástica basado en el Simulated Annealing. En cada caso se han obtenido resultados en forma de modelos analíticos de rendimiento sencillos.

Se han analizado las operaciones de comunicación punto a punto y colectivas de la librería de pase de mensajes MPICH-G2 utilizada por el proyecto CrossGrid para obtener un modelo analítico de los tiempos de ejecución de las comunicaciones. Se ha desarrollado para ello un modelo simplificado de alto nivel para las comunicaciones punto a punto y una matriz de coeficientes para caracterizar los parámetros de red efectivos en tiempo real de las comunicaciones entre cada posible pareja de nodos. El modelo ha sido incorporado al middleware de CrossGrid a través de un interfaz gráfico, permitiendo a los usuarios determinar el comportamiento de las comunicaciones dinámicamente sobre el sistema en función de su situación actual en términos de la carga de los nodos y de la red.

Se han examinado los métodos iterativos implementados por la librería de álgebra lineal dispersa PARAISO. Se han identificado las operaciones presentes en las redistribuciones de los datos matriciales entre los formatos de datos intrínsecos de HPF y de la implementación en MPI del producto matriz dispersa-vector. Se ha desarrollado un modelo de computaciones con expresiones analíticas para todos los métodos y precondicionadores presentes en la librería. Así mismo, se han modelado analíticamente las principales comunicaciones colectivas involucradas.

Se ha estudiado el comportamiento de una herramienta de simulación de la dispersión atmosférica de contaminantes basada en la versión paralela del código STEM-II. Se ha identificado la rutina que consume la mayor parte del tiempo de ejecución. A partir del estudio de un amplio conjunto de parámetros de entrada, se ha obtenido un modelo analítico para las computaciones y las comunicaciones que requiere dicha rutina. Es de especial relevancia el uso de estos modelos para abordar el problema del balanceo de la carga computacional entre

los procesadores en la ejecución de la aplicación. El modelo se ha incorporado al *middleware* de CrossGrid junto a un interfaz gráfico para un uso cómodo.

Se ha analizado un algoritmo de optimización basado en el Simulated Annealing presente en la herramienta RULES en el campo de la planificación territorial. Se han caracterizado estadísticamente las ejecuciones de la versión secuencial del algoritmo para un amplio conjunto de parámetros de entrada y obtenido un modelo analítico de computaciones tanto para las operaciones enteras como para las operaciones en punto flotante, mediante la herramienta TIA, para la exploración de dependencias en términos y selección del modelo final. Se ha extendido el algoritmo para su ejecución paralela y desarrollado un modelo analítico para las comunicaciones en las varias alternativas propuestas. El modelo permite determinar las mejores propuestas y elegir su mejor parametrización.

Publicaciones

Las siguientes publicaciones son resultado, directo o indirecto, de las investigaciones realizadas para este trabajo:

- *Predicción de rendementos de códigos irregulares no AP3000*; D. Blanco, V. Blanco, M. Boullón, J.C. Cabaleiro, T.F. Pena, J.J. Pombo, F.F. Rivera; Revista Díxitos, Santiago de Compostela; pp. 4 - 5; 1999. Es un texto de divulgación realizada en una publicación del CESGA.

- *Algorithm based on simulated annealing for land use allocation*; I. Santé, M. Boullón, R. Crecente, D. Miranda; Computers & Geosciences; vol 34(3); pp. 259 - 268; 2008

- *Optimising land use allocation at municipal level by combining multicriteria evaluation and linear programming*; I. Santé, R. Crecente, M. Boullón, D. Miranda; Spatial Decision Support for Urban and Environmental Planning. A Collection of Case Studies. ISBN 978-983-3718-53-5., Malaysia; pp. 33 - 60; 2009

- *Migración de una aplicación MPI a una plataforma Grid*; M.J. Martín, P. González, J.C. Mouriño, R. Doallo, M. Boullón, F.F. Rivera; XIII Jornadas de Paralelismo, Lleida, España; 2002

- *A Grid enabled air quality simulation*; J.C. Mouriño, M.J. Martín, P. González, M. Boullón, J.C. Cabaleiro, T.F. Pena, F.F. Rivera, R. Doallo; First European Across Grids Conference, Santiago de Compostela, España; 2003

- *Mejora de la localidad en SMPs: el producto matriz dispersa-vector como caso de estudio*; J.C. Pichel, D.B. Heras, J.C. Cabaleiro, M. Boullón, F.F. Rivera; XV Jornadas de Paralelismo, Almería, España; 2004

- *Modelling execution time of selected computation and communication kernels on Grid*; M. Boullón, J.C. Cabaleiro, R. Doallo, P. González, D.R. Martínez, M.J. Martín, J.C. Mouriño, T.F. Pena, F.F. Rivera; European Grid Conference (EGC2005), Amsterdam, The Nederlands; 2005

- *Predicción de rendimiento de una aplicación Grid para el control de inundaciones*; J.L. Albín, D.R. Martínez, M. Boullón, J.C. Cabaleiro, T.F. Pena, F.F. Rivera; XVI Jornadas de Paralelismo, Granada, España; 2005

- *Diseño de un algoritmo basado en el simulated annealing para la generación de escenarios de uso del suelo*; I. Santé, R. Crecente, M. Boullón; III Congreso Internacional Ciudad y Territorio Virtual, Bilbao, España; 2006

- *RULES. Sistema de Ayuda a la Decisión Espacial para la planificación de los usos del suelo rural.*; Inés Santé Riveira, Rafael Crecente Maseda, Marcos Boullón Magán; SC-407-6; 2006

- *Analytical Performance Models of Parallel Programs in Clusters*; Diego R. Martínez, Vicente Blanco, Marcos Boullón, José C. Cabaleiro, Tomás F. Pena; Parallel Computing: Architectures, Algorithms and Applications, ParCo 2007, Forschungszentrum Jülich and RWTH Aachen University, Germany, 4-7 September 2007, Germany; 2007

- *A parallel algorithm based on simulated annealing for land use zoning plans*; M. Suárez, I. Santé, F.F. Rivera, R. Crecente, M. Boullón, J. Porta, J. Parapar, R. Doallo; The 2011 International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, EEUU; 2011

Bibliografía

[1] *EGEE: The Enabling Grids for E-science in Europe*. http://www.eu-egee.org.

[2] *WSRF – The WS-Resource Framework*. http://www.globus.org/wsrf.

[3] *OpenMP API specification for parallel programming*, 1997. http://www.openmp.org.

[4] *PCL - The Performance Counter Library*, 2003. http://berrendorf.inf.h-brs.de/PCL/PCL.html.

[5] *CrossGrid project details*, 2005. http://cordis.europa.eu/project/rcn/63588_en.html.

[6] *HPCTOOLKIT*, 2011. http://www.hpctoolkit.org.

[7] *Performance Application Programming Interface. PAPI user's guide, version 2.3*, 2011. http://icl.cs.utk.edu/papi.

[8] *The Prophesy System Website*, 2011. http://prophesy.cs.tamu.edu.

[9] *SCALASCA*, 2011. http://www.scalasca.org.

[10] *Top 500 Supercomputer Sites*, 2015. http://top500.org/statistics/list/.

[11] Ahmar Abbas. *Grid computing. A practical guide to technology and applications*. Charles River Media, 2004.

[12] David Abramson, Mohan Krishna Amoorthy, and Henry Dang. Simulated annealing cooling schedules for the school timetabling problem. *Asia-Pacific Journal of Operational Research*, 16(1):1, 1999.

[13] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R. Tallent. Hpctoolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

[14] Vikram S. Adve, Rajive Bagrodia, James C. Browne, Ewa Deelman, Aditya Dube, Elias Houstis, John Rice, Rizos Sakellariou, David Sundaram-stukel, Patricia J. Teller, and Mary K. Vernon. POEMS: End-to-end performance design of large parallel adaptive computational systems. *IEEE Transactions on Software Engineering*, 26:1027–1048, 2001.

[15] Jeroen C.J.H. Aerts and Gerard B.M. Heuvelink. Using simulated annealing for resource allocation. *International Journal of Geographical Information Science*, 16(6):571–587, 2002.

[16] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. LogGP: Incorporating long messages into the LogP model for parallel computation. *Journal of Parallel and Distributed Computing*, 44:71–79, 1997.

[17] R.J. Allan, Y.F. Hu, and P. Lockey. A survey of parallel numerical analysis software: 2nd edition. Parallel application software on High-Performance Computers. Technical Report DL-TR-99-01, CLRC Daresbury Laboratory, 1999.

[18] J. Almond and D. Snelling. UNICORE: uniform access to supercomputing as an element of electronic comerce. *Future Generation Computer Systems, FGCS*, 15(5–6):539–548, 1999.

[19] G. Amdahl. Validity of the single-processor approach to achieving large-scale computing capabilities. *Proc. 1967 AFIPS Conf*, 30:483, 1967.

[20] Wolfram Amme, Peter Braun, Welf Löwe, and Eberhard Zehendner. LogP modelling of list algorithms. *11th Symposium on Computer Architecture and High Performance Computing*, 1999.

[21] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of Cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[22] Ruth A. Aydt. The Pablo self-defining data format. Technical report, Computer Science Dept. University of Illinois, Urbana-Champaign, IL, USA, 1992.

[23] D. Bailey, E. Barscz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS parallel benchmarks. Technical Report RNR-94-007, NASA, Ames Research Center, 1994.

[24] Amotz Bar-Noy and Shlomo Kipnis. Designing broadcasting algorithms in the postal model for message–passing systems. *Mathematical Systems Theory*, 27(5):431–452, 1994.

[25] Ingrid Barcena, José Antonio Becerra, Joan Cambras, Richard Duro, Carlos Fernández, Javier Fontán, Andrés Gómez, Ignacio López, Caterina Parals, José Carlos Pérez, and Juan Villasuso. A Grid supercomputing enviroment for high demand computational applications. Technical report, Centre de Supercomputació de Catalunya (CESCA), Autonomous Systems Group (GSA), University of A Coruña (UDC), Centro de Supercomputación de Galicia (CESGA), 2000. http://www.cesga.es/pdf/Grid_CESGA_CESCA.PDF.

[26] M. Barnett, S. Gupta, D. G. Payne, L. Shuler, Robert van de Geijn, and Jerrell Watts. Interprocessor collective communication library (InterCom). *Scalable High-Performance Computing Conference, Knoxville, Tennessee. IEEE Computer Society Press*, pages 357–364, 1994.

[27] M. Barnett, S. Gupta, D.G. Payne, L. Shuler, Robert van de Geijn, and Jerrell Watts. Building a high-performance collective communication library. *Supercomputing'94, Supercomputing, Washington, DC, USA. IEEE Computer Society Press*, pages 107–116, 1994.

[28] R. Barret, M. Berry, et al. Templates for the solution of linear systems: Building blocks for iterative methods. *SIAM*, 1994.

[29] C. Bell, D. Bonachea, Y. Cote, J. Duell, P. Hargrove, P. Husbands, C. Iancu, M. Welcome, and K. Yelick. An evaluation of current high-performance networks. *Proc. International Parallel and Distributed Processing Symposium*, 2003.

[30] Gordon Bell and Jim Gray. What's next in high-performance computing? *Communications of the ACM*, 45(2):91–95, 2002.

[31] Fran Berman, Geoffrey Fox, and Tony Hey. *Grid Computing*. John Wiley & Sons, Ltd, 2003.

[32] Rudolf Berrendorf and Bernd Mohr. Pcl - the performance counter library. a common interface to access hardware performance counters on microprocessors. Technical report, University of Applied Sciences Bonn-Rhein-Sieg and Central Institute for Applied Mathematics at Research Centre Julich, 2002.

[33] Dimitris Bertsimas, John Tsitsiklis, et al. Simulated annealing. *Statistical science*, 8(1):10–15, 1993.

[34] V. Blanco, J.C. Cabaleiro, P. Gonzalez, D.B. Heras, T.F. Pena, J.J. Pombo, and F.F. Rivera. A performance analysis tool for irregular codes in HPF. Technical report, Dept. Electronics and Computer Science. Univ. Santiago de Compostela, 2000.

[35] V. Blanco, J.C. Cabaleiro, P. González, D.B. Heras, T.F. Pena, J. J. Pombo, and F. F. Rivera. HPI (HPF & MPI hybrid) implementation of Paraiso. Technical report, Dept. Electronics and Computer Science. Univ. Santiago de Compostela, 2000.

[36] V. Blanco, J.C. Cabaleiro, P. González, D.B. Heras, T.F. Pena, J.J. Pombo, and F.F. Rivera. A performance analysis tool for irregular codes in HPF. *Fifth European SGI/Cray MPP Workshop, Bologna, Italy, CINECA/SGI*, 1999.

[37] V. Blanco, J.C. Cabaleiro, P. González, D.B. Heras, T.F. Pena, J.J. Pombo, and F.F. Rivera. HPF implementation of Paraiso. Technical report, Dept. Electronics and Computer Science. Univ. Santiago de Compostela, 2000.

[38] V. Blanco, J.C. Cabaleiro, P. González, D.B. Heras, T.F. Pena, J.J. Pombo, and F.F. Rivera. Performance of parallel iterative solvers: a library, a prediction model, and a visualization tool. *Journal of Information Science and Engineering. Special Issue: Parallel and Distributed Computing*, 8(5):763–785, 2002.

[39] V. Blanco, J.C. Cabaleiro, P. González, D.B. Heras, T.F. Pena, J.J. Pombo, and F.F. Rivera. Performance prediction for parallel iterative solvers. In Peter M.A. Sloot, Alfons G. Hoekstra, C.J. Kenneth Tan, and Jack J. Dongarra, editors, *Computational*

Science - ICCS 2002, volume 2330 of *Lecture Notes in Computer Science*, pages 923–932. Springer Berlin Heidelberg, 2002.

[40] V. Blanco, J.C. Cabaleiro, P. González, D.B. Heras, T.F. Pena, J.J. Pombo, and F.F. Rivera. A performance visualization tool for HPF and MPI iterative solvers. *16th International Parallel and Distributed Processing Symposium (IPDPS'02). Workshop on Parallel and Distributed Scientific and Engineering Computing with Applications (PDSECA'02), Marriot Marina, Fort Lauderdale, Florida, USA. IEEE Computer Society*, page 235, 2002.

[41] V. Blanco, J. A. González, C. León, C. Rodríguez, G. Rodríguez, and M. Printista. Predicting the performance of parallel programs. *Parallel Computing*, 30:337–356, 2004.

[42] V. Blanco, P. Gonzalez J.C. Cabaleiro and, D.B. Heras, T.F. Pena, J.J. Pombo, and F.F. Rivera. HPI (HPF & MPI hybrid) implementation of Paraiso. *SIAM*, 1994.

[43] François Bodin, Peter Beckman, Dennis Gannon, Srinivas Narayana, and Shelby X. Yang. Distributed pC++ basic ideas for an object parallel language. *Scientific Programming*, 2(3):7–22, 1993.

[44] J.L. Bosque and L.P. Perez. HLogGP: a new parallel computational model for heterogeneous clusters. *Proc. IEEE International Symposium on Cluster Computing and the Grid, CCGrid*, 2004.

[45] M. Boullón, J.C. Cabaleiro, R. Doallo, P. González, D.R. Martínez, M. Martín, J.C. Mouriño, T. F. Pena, and F.F. Rivera. Modeling execution time of selected computation and communication kernels on grids. In *Proceedings of the 2005 European Conference on Advances in Grid Computing*, EGC'05, pages 731–740, Berlin, Heidelberg, 2005. Springer-Verlag.

[46] Richard K. Brail and Richard E. Klosterman. *Planning support systems: integrating geographic information systems, models, and visualization tools*. ESRI, Inc., 2001.

[47] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *International Journal of High-Performance Computing Applications*, 14(3):189–204, 2000.

[48] Marian Bubak, Maciej Malawski, and Katarzyna Zajac. The CrossGrid architecture: Applications, tools, and grid services. In Francisco Fernández Rivera, Marian Bubak, Andrés Gómez Tato, and Ramón Doallo, editors, *Grid Computing*, volume 2970 of *Lecture Notes in Computer Science*, pages 309–316. Springer Berlin Heidelberg, 2004.

[49] Marian Bubak, Jesús Marco, Holger Marten, Norbert Meyer, Marian Noga, Peter A.M. Sloot, and Michal Turala. *CrossGrid - Development of grid Environment for Interactive Applications*. http://www.crossgrid.org.

[50] Marian Bubak, Piotr Nowakowski, and Robert Pajak. An overview of european grid projects. In Francisco Fernández Rivera, Marian Bubak, Andrés Gómez Tato, and Ramón Doallo, editors, *Grid Computing*, volume 2970 of *Lecture Notes in Computer Science*, pages 299–308. Springer Berlin Heidelberg, 2004.

[51] B. Buck and J. Hollingsworth. An API for runtime code patching. *International Journal of High Performance Computing Applications*, 14(4):317–329, 2000.

[52] Rajkumar Buyya, editor. *High Performance Cluster Computing: Architectures and Systems*. Prentice Hall PTR, 1999.

[53] K.W. Cameron, R. Ge, and X.H. Sun. lognP and log3P: Accurate analytical models of point-to-point communication in distributed systems. *IEEE Transactions on Computers*, 56(3):314–327, 2007.

[54] José I. Barredo Cano. *Sistemas de información geográfica y evaluación multicriterio: en la ordenación del territorio*. Ra-Ma, Madrid, 1996.

[55] G.R. Carmichael and L.K. Peters. An eulerian transport/transformation/removal model for SO2 and Sulfate-I. *Atmospheric Environment*, 18:937–952, 1984.

[56] G.R. Carmichael and L.K. Peters. An eulerian transport/transformation/removal model for SO2 and sulfate-II. model calculation of SOx transport in the Eastern United States. *Atmospheric Environment*, 18:953–967, 1984.

[57] G.R. Carmichael, L.K. Peters, and R.D. Saylor. The STEM-II regional scale acid deposition and photochemical oxidant model-I. an overview of model development and applications. *Atmospheric Environment*, 25:2077–2090, 1991.

[58] Michael Carton and Alvin Despain. New directions in scalable shared–memory multiprocessor architectures. *IEEE Computer Society*, 23(6):71–83, 1990.

[59] Vladimír Černỳ. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of optimization theory and applications*, 45(1):41–51, 1985.

[60] David Chaiken, Craig Fields, Kiyoshi Kurihara, and Anant Agarwal. Directory–based cache coherence in large–scale multiprocessors. *IEEE Computer Society*, 23(6):49–58, 1990.

[61] Rohit Chandra, Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan, and Jeff McDonald. *Parallel Programming in OpenMP*. Elsevier, Inc., 2000.

[62] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, and Kevin Skadron. A performance study of general-purpose applications on graphics processors using CUDA. *Journal of Parallel and Distributed Computing*, 68(10):1370–1380, 2008.

[63] J. Chen and V. Taylor. Parapart: Parallel mesh partitioning for efficient use of distributed systems. *Concurrency: Practice and Experience*, 12:111–123, 2000.

[64] Jared L. Cohon. *Multiobjective Programming and Planning*, volume 140. Courier Corporation, 2004.

[65] George Coulouris, Jean Dollimore, and Tim Kindberg. *Sistemas distribuidos. Conceptos y diseño*. Addison-Wesley, 2001.

[66] David Culler, Lok Tin Liu, Richard P. Martin, and Chad Yoshikawa. Assesing fast network interfaces. *IEEE Micro*, 16(1):35–43, 1996.

[67] David E. Culler and Jaswinder Pal Singh. *Parallel Computer Architecture. A Hardware/Software Approach*. Morgan Kaufmann Publishers, Inc, 1999.

[68] D.E. Culler, R.M. Karp, D.A. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. *Principles Practice of Parallel Programming*, 68(10):1370–1380, 1993.

[69] Karl Czajkowski, Ian Foster, Nick Karonis, Carl Kesselman, Stuart Martin, Warren Smith, and Steven Tuecke. A resource management architecture for metacomputing systems. In DrorG. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1459 of *Lecture Notes in Computer Science*, pages 62–82. Springer Berlin Heidelberg, 1998.

[70] Karl Czajkowski, Ian Foster, and Carl Kesselman. Resource co-allocation in computational grids. In *In Proceedings Of The Eighth IEEE International Symposium On High Performance Distributed Computing (HPDC-8)*, pages 219–228. IEEE Computer Society, 1999.

[71] Centro Europeo de Paralelismo de Barcelona. *Paraver*. http://www.cepba.upc.edu/paraver.

[72] E. de Sturler and D. Loher. Parallel solution of irregular, sparse matrix problems using High Performance Fortran. Technical Report TR-96-39, Swiss Center for Scientific Computing, 1996.

[73] Bronis R De Supinski and Nicholas T Karonis. Accurately measuring MPI broadcasts in a computational grid. In *High Performance Distributed Computing, 1999. Proceedings. The Eighth International Symposium on*, pages 29–37. IEEE, 1999.

[74] Ma. del Rosario Méndez. *Análisis y evaluación de un modelo de calidad del aire para la estimación de la deposición ácida*. PhD thesis, Universidade de Santiago de Compostela, 2001.

[75] Wolfgang E. Denzel, Jian Li, Peter Walker, and Yuho Jin. A framework for end-to-end simulation of high-performance computing systems. *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, 2008.

[76] K. Dincer, K.A. Hawick, A. Choudary, and G.C. Fox. High Performance Fortran and possible extensions to support conjugate gradient algorithms. Technical Report SCCS 703, Northeast Parallel Architectures Center, Syracuse, NY, 1995.

[77] S. Dong, G.E. Karniadakes, and N.T. Karonis. Cross-site computations on the TeraGrid. *Computing in Science Engineering*, 7(5):14–23, 2005.

[78] Jack Dongarra, Piotr Luszczek, and Antoine Petitet. The LINPACK Benchmark: Past, present and future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820, 2004.

[79] Jack Dongarra, Allen D. Malony, Shirley Moore, Philip Mucci, and Sameer Shende. Performance instrumentation and measurement for terascale systems. *Proceedings of the 2003 international conference on Computational science, ICCS*, 2003.

[80] Jack Dongarra, Thomas Sterling, Horst Simon, and Erich Strohmaier. High-Performance Computing: Clusters, constellations, MPPs, and future directions. *Computing in Science and Engineering*, 7:51–59, 2005.

[81] José Duato, Sudhakar Yalamanchili, and Lionel Li. Interconnection networks: An engineering approach. *IEEE Computer Society*, 1997.

[82] Andrea C. Dusseau, David E. Culler, Klaus Erik Schause, and Richard P. Martin. Fast parallel sorting under LogP. experience with the CM–5. *IEEE Transactions on Parallel and Distributed Systems*, 7(8):791–805, 1996.

[83] J. Ronald Eastman, Hong Jiang, and James Toledano. Multi-criteria and multi-objective decision making for land allocation using GIS. In *Multicriteria analysis for land-use management*, pages 227–251. Springer, 1998.

[84] John W. Eaton, David Bateman, and Sören Hauberg. *GNU Octave version 3.0.1 manual: a high-level interactive language for numerical computations*. CreateSpace Independent Publishing Platform, 2009. ISBN 1441413006.

[85] R.W. Eglese. Simulated annealing: a tool for operational research. *European journal of operational research*, 46(3):271–281, 1990.

[86] Victor Eijkhout. Overview of iterative linear system solver packages. *NHSE Review*, 3(1), 1998.

[87] Joern Eisenbiegler, Welf Loewe, and Andreas Wehrenpfennig. On the optimization by redundancy using an extended LogP model. *APDC'97: Proceedings of the 1997 Advances in Parallel and Distributed Computing Conference*, 1997.

[88] M.A. Saleh Elmohamed, Paul Coddington, and Geoffrey Fox. A comparison of annealing techniques for academic course scheduling. In *Practice and Theory of Automated Timetabling II*, pages 92–112. Springer, 1998.

[89] R. Elsasser, B. Monien, and R. Preis. Diffusive load balancing schemes for heterogeneous networks. *Proc. SPAA'2000, Maine*, 2000.

[90] C. Evangelinos and C.N. Hill. Cloud computing for parallel scientific HPC applications: Feasibility of running coupled atmosphere-ocean climate models on amazon's ec2. *Proc. Cloud Computing and Its Applications*, 2008.

[91] FAO. *A Framework for Land Evaluation*. FAO, Rome, 1976.

[92] John T. Feo, David C. Cann, and Rodney R. Oldehoeft. A report on the Sisal language project. *Journal of Parallel and Distributed Computing*, 10(4):349–366, 1990.

[93] M.A. Fleischer and S.H. Jacobson. Scale invariance properties in the simulated annealing algorithm. *Methodology and Computing in Applied Probability*, 4(3):219–241, 2002.

[94] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21(9):948–960, 1972.

[95] Steven Fortune and James Wyllie. Parallelism in random access machines. *Tenth Annual ACM Symp. Theory of Computing*, 1978.

[96] HPF Forum. *High Performance Fortran Language Specification*, 1997. http://dacnet.rice.edu/Depts/CRPC/HPFF/index.cfm.

[97] MPI Forum. MPI: A message-passing interface standard. *The International Journal of Supercomputer Applications and High Performance Computing*, 8:159–416, 1994.

[98] I. Foster and C. Kesselman. The Globus project: A status report. *Future Generation Computer Systems*, 15(5), 1999.

[99] I. Foster, C. Kesselman, J.M. Nick, and S. Tuecke. Grid services for distributed system integration. *IEEE Computer Society*, 35(6):37–46, 2002.

[100] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *The International Journal of High Performance Computing Applications*, 15(3):200–222, 2001.

[101] Ian Foster, Jonathan Geisler, William Gropp, Nicholas Karonis, Ewing Lusk, George Thiruvathukal, and Steven Tuecke. Wide-area implementation of the message passing interface. *Parallel Computing*, 24:1735–1749, 1998.

[102] Ian Foster and Nicholas T. Karonis. A grid-enabled MPI: Message passing in heterogeneous distributed computing systems. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, SC'98, pages 1–11, Washington, DC, USA, 1998. IEEE Computer Society.

[103] Ian Foster and Carl Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Inc., 1999.

[104] Ian Foster and Carl Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc, 2003.

[105] Ian Foster and Carl Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Elsevier, Inc, 2nd edition, 2004.

[106] Ian T. Foster, Robert Olson, and Steven Tuecke. *Programming in Fortran M*. Argonne National Laboratory, 1993.

[107] Geoffrey Fox, Seema Hiranandani, Ken Kennedy, Charles Koelbel, Ulrich Kremer, Chau-Wen Tseng, and Min-You Wu. Fortran D language specification. Technical report, 1990.

[108] J.C. Mouriño Gallego. *Aplicación de la computación de altas prestaciones al modelado de la calidad del aire*. PhD thesis, Universidade da Coruña, 2006.

[109] Luisa Gargano and Adele A. Rescigno. Fast collective communication by packets in the postal model. *Networks*, 31:67–79, 1998.

[110] Chao-Yang Gau and Mark A. Stadtherr. Parallel interval-newton using message passing: Dynamic load balancing strategies. *Proc. SC2001, Denver, ACM*, 2001.

[111] Stan Geertman and John Stillwell. *Planning Support Systems in Practice*. Springer Science & Business Media, 2003.

[112] Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The SCALASCA performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, 2010.

[113] Saul B. Gelfand and Sanjoy K. Mitter. Simulated annealing type algorithms for multivariate optimization. *Algorithmica*, 6(1-6):419–436, 1991.

[114] Stuart Geman and Donald Geman. Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 6:721–741, 1984.

[115] David Geneletti and Alias Abdullah. *Spatial Decision Support for Urban and Environmental Planning. A collection os case studies*. Arah Publications, 2009.

[116] A. Ghiselli. DataGrid prototype 1. *TERENA Networking Conference*, 2002.

[117] A. Goicoechea, D.R. Hansen, and L. Duckstein. *Multiobjective decision analysis with engineering and business applications*. John Wilet & Sons, USA, 1982.

[118] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. *Introduction to Parallel Computing*. Addison-Wesley, 2003.

[119] Andrew S. Grimshaw, Wm A. Wulf, and The Legion Team. The Legion vision of a worldwide virtual computer. *Commun. ACM*, 40(1):39–45, 1997.

[120] Eric Grobelny, David Bueno, Ian Troxel, Alan D. George, and Jeffrey S. Vetter. FASE: A framework for scalable performance prediction of HPC systems and applications. *Simulation*, 83(10):721–745, 2007.

[121] 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.

[122] S.D. Hammond, G.R. Mudalige, J.A. Smith, S.A. Jarvis, J.A. Herdman, and A. Vadgama. WARPP: A toolkit for simulating high performance parallel scientific codes. *SIMUTools09*, 2009.

[123] S.D. Hammond, J.A. Smith, G.R. Mudalige, and S.A. Jarvis. Predictive simulation of HPC applications. *IEEE 23rd International Conference on Advanced Information Networking and Applications, AINA*, 2009.

[124] Britton Harris. Urban development models: New tools for planning. *Journal of the American Institute of Planners*, 31(2):90–95, 1965.

[125] Darrall Henderson, Sheldon Jacobson H., and Alan W. Johnson. The theory and practice of simulated annealing. In *Handbook of metaheuristics*, pages 287–319. Springer, 2003.

[126] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2006.

[127] W. Daniel Hillis and Jr. Guy L. Steele. Data parallel algorithms. *Communications of the ACM*, 1986.

[128] Susan Hinrichs, Corey Kosak, David R. O'Hallaron, Thomas Stricker, and Riichiro Take. An architecture for optimal all-to-all personalized communication. *ACM Symposium on Parallel Algorithms and Architectures*, pages 310–319, 1994.

[129] R. Hockney. The communication challenge for MPP: Intel Paragon and Meiko CS-2. *Parallel Computing*, 20:389–398, 2004.

[130] T. Hoefler, T. Schneider, and A. Lumsdaine. LogGP in theory and practice — an in-depth analysis of modern interconnection networks and benchmarking methods for collective operations. *Simulation Modelling Practice and Theory*, 17(9):1511–1521, 2009.

[131] T. Howes. *Understanding and Deploying LDAP Directory Services*. Addison-Wesley Professional, 2002.

[132] Daniel C. Hyde. Introduction to the programming language occam. *Department of Computer Science Bucknell University, Lewisburg*, 1995.

[133] G. Iannello. Efficient algorithms for the reduce-scatter operation in LogGP. *IEEE Transactions on Parallel and Distributed Systems*, 8(9):970–982, 1997.

[134] Lester Ingber. Very fast simulated re-annealing. *Mathematical and computer modelling*, 12(8):967–973, 1989.

[135] Lester Ingber. Simulated annealing: Practice versus theory. *Mathematical and computer modelling*, 18(11):29–57, 1993.

[136] R.K. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons, Ltd, 1991.

[137] Alan W. Johnson, Darrall Henderson, and Sheldon H. Jacobson. The rise and fall of simulated annealing. 2003.

[138] Richard P. Larowe Jr and Carla Schlatter Ellis. Experimental comparison of memory management policies for NUMA multiprocessors. *ACM Transactions on Computer Systems*, 9(4):319–363, 1991.

[139] Alcino Dall'Igna Júnior, Renato S. Silva, Kleber C. Mundim, and Laurent E. Dardenne. Performance and parameterization of the algorithm Simplified Generalized Simulated Annealing. *Genetics and Molecular Biology*, 27(4):616–622, 2004.

[140] N. Karonis, B. Toonen, and I. Foster. MPICH-G2: A grid-enabled implementation of the message passing interface. *Journal of Parallel and Distributed Computing (JPDC)*, 63(5):551–560, 2003.

[141] N.T. Karonis, B.R. de Supinski, I. Foster, W. Gropp, E. Lusk, and J. Bresnahan. Exploiting hierarchy in parallel computer networks to optimize collective operation performance. In *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, pages 377–384, 2000.

[142] Richard M. Karp, Abhijit Sahay, Eunice E. Santos, and Klaus E. Schauser. Optimal broadcast and summation in the LogP model. *ACM Symposium on Parallel Algorithms and Architectures*, pages 142–153, 1993.

[143] R.M. Karp and V. Ramachandran. *HandBook of Theoretical Computer Science*, volume A. Elsevier Science. MIT Press, 1990. chapter Parallel algorithms for shared-memory machines.

[144] Thilo Kielmann, Henri E. Bal, and Sergei Gorlatch. Bandwidth-efficient collective communication for clustered wide area systems. In *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, pages 492–499. IEEE, 2000.

[145] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *Science*, 4598:671–680, 1983.

[146] Richard E. Klosterman. Evolving views of computer-aided planning. *Journal of Planning Literature*, 6(3):249–260, 1992.

[147] Richard E. Klosterman. Planning support systems: a new perspective on computer-aided planning. *Journal of Planning education and research*, 17(1):45–54, 1997.

[148] M. Kruczowski, B. Palak, M. Plociennik, P. Wolniewicz, M. Kupczyk, N. Meyer, S. Beco, and Y. Perros. Roaming access and portals: software requirements specification. Technical report, EU-Crossgrid Project Technical Report, 2002. www.eu-crossgrid.org/Deliverables/M3pdf/SRS_TASK_3-1.pdf.

[149] V. Kumar. *Introduction to parallel computing : design and analysis of algorithms*. Benjamin/Cummings Pub., Co, Redwood City, Calif. USA, 1994.

[150] Miroslaw Kupczyk, Rafal Lichwala, Norbert Meyer, Bartosz Palak, Marcin Plóciennik, Maciej Stroinski, and Pawel Wolniewicz. The Migrating Desktop as a GUI framework for the application-on-demand concept. In Marian Bubak, GeertDick van Albada, Peter M.A. Sloot, and Jack J. Dongarra, editors, *Computational Science - ICCS 2004*, volume 3036 of *Lecture Notes in Computer Science*, pages 91–98. Springer Berlin Heidelberg, 2004.

[151] Jesús Labarta, Sergi Girona, Vincent Pillet, Toni Cortés, and Luis Gregoris. Dip : A parallel program development environment. *2nd International EuroPar Conference (EuroPar 96), Lyon (France)*, 1996.

[152] Zhiling Lan, Valerie E. Taylor, and Greg Bryan. Dynamic load balancing of SAMR applications on distributed systems. *Proc. SC2001, Denver, ACM*, 2001.

[153] Alexey Lastovetsky, Vladimir Rychkov, and Maureen O'Flynn. MPIBlib: Benchmarking MPI communications for parallel computing on homogeneous and heterogeneous clusters. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 227–238, 2008.

[154] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufman Publishers, San Mateo, CA, USA, 1992.

[155] Wei Li and Keshav Pingali. Access normalization: Loop restructuring for NUMA compilers. *In Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V) ACM Sigplan Notices*, 27:285–295, 1992.

[156] Kathleen A. Lindlan, Janice Cuny, Allen D. Malony, Sameer Shende, Reid Rivenburgh, Craig Rasmussen, and Bernd Mohr. A tool framework for static and dynamic analysis of object-oriented software with templates. *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, 2000.

[157] Marco Locatelli. Simulated annealing algorithms for continuous global optimization. In *Handbook of global optimization*, pages 179–229. Springer, 2002.

[158] Thomas Ludwig and Roland Wismüller. OMIS 2.0 - a universal interface for monitoring systems. In *Proceedings of 4th European PVM/MPI Users' Group Meeting*, pages 267–276. Springer Verlag, 1997.

[159] David Luebke, Mark Harris, Jens Krüger, Tim Purcell, Naga Govindaraju, Ian Buck, Cliff Woolley, and Aaron Lefohn. GPGPU: general purpose computation on graphics hardware. *SIGGRAPH'04: ACM SIGGRAPH 2004 Course Notes*, 2004.

[160] María J. Martín, David E. Singh, J. Carlos Mouriño, Francisco F. Rivera, Ramón Doallo, and Javier D. Bruguera. High performance air pollution modeling for a power plant environment. *Parallel Computing*, 2003.

[161] Diego R. Martínez. Modelización y mejora del comportamento computacional en la simulación paralela de la dispersión atmosférica de contaminantes. Master's thesis, Universidad de Santiago de Compostela, 2005.

[162] Diego R. Martínez. *Modelado analítico del rendimiento de aplicaciones en sistemas paralelos*. PhD thesis, Universidad de Santiago de Compostela, 2011.

[163] The MathWorks, Inc., Natick, Massachusetts, United States. *MATLAB and Statistics Toolbox Release 2012b*, 2012.

[164] P.K. McKinley, Y.J. Tsai, and D. Robinson. Collective communication in wormhole–routed massively parallel computers. *IEEE Computer Society*, 1995.

[165] Guillermo A. Mendoza. GIS-based multicriteria approaches to land use suitability assessment and allocation. *United States Department Of Agriculture Forest Service General Technical Report NC*, pages 89–94, 2000.

[166] Nicholas Metropolis, Arianna Rosenbluth W, Marshall Rosenbluth N, Augusta Teller H, and Edward Teller. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21(6):1087–1092, 1953.

[167] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The Paradyn parallel performance measurement tools. *IEEE Computer Society*, 1995.

[168] Prasenjit Mitra, David Payne, Lance Shuler, Robert van de Geijn, and Jerrell Watts. Fast collective communication libraries, please. *Proceedings of the Intel Supercomputing Users' Group Meeting*, 1995.

[169] Bernd Mohr, Allen D. Malony, Sameer Shende, and Felix Wolf. Towards a performance tool interface for OpenMP: an approach based on directive rewriting. *Proceedings of the Third European Workshop on OpenMP*, 2001.

[170] J.C. Mouriño, D.E. Singh, M.J. Martín, J.M. Eiroa, F.F. Rivera, R. Doallo, and J.D. Bruguera. Parallelization of the STEM-II air quality model. *Proc. Int. Symp. High Performance Computing and Networking (HPCN2001), Amsterdam*, 2001.

[171] J.C. Mouriño, D.E. Singh, M.J. Martín, F.F. Rivera, R. Doallo, and J.D. Bruguera. The STEM-II air quality model on a distributed memory system. *Workshop on High Performance Scientific and Engineering computing with Applications (HPSECA-2001), Proceedings of the 2001 ICPP Workshops, Valencia*, pages 85–92, 2001.

[172] W.E. Nagel, A. Arnold, M. Weber, H.Ch. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12:69–80, 1996.

[173] Aroon Nataraj, Allen D. Malony, Alan Morris, Dorian C. Arnold, and Barton P. Miller. A framework for scalable, parallel performance monitoring. *Concurrency and Computation: Practice and Experience*, 22(6):720–735, 2010.

[174] L.M. Ni. Issues in designing truly scalable interconection networks. *1996 ICPP Workshop on Challenges for Parallel Processing*, pages 74–83, 1996.

[175] G.R. Nudd, D.J. Kerbyson, E. Papaefstathiou, S.C. Perry, J.S. Harper, and D.V. Wilcox. Pace–a toolset for the performance prediction of parallel and distributed systems. *Int. J. High Perform. Comput. Appl.*, 14(3):228–251, August 2000.

[176] Ariel Oleksiak and Jarek Nabrzyski. Comparison of grid middleware in european grid projects. In Francisco Fernández Rivera, Marian Bubak, Andrés Gómez Tato, and Ramón Doallo, editors, *Grid Computing*, volume 2970 of *Lecture Notes in Computer Science*, pages 317–325. Springer Berlin Heidelberg, 2004.

[177] L. Oliker and R. Biswas. PLUM: parallel load balancing for adaptive refined meshes. *Journal of Parallel and Distributed Computing*, 47(2):109–124, 1997.

[178] Esin Onbaşoğlu and Linet Özdamar. Parallel simulated annealing algorithms in global optimization. *Journal of Global Optimization*, 19(1):27–50, 2001.

[179] P.S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann Publishers, Inc., 2000.

[180] Michael Pahud. Ip3t, un outil de prédiction de performance pour applications parallèles irrégulières. Technical report, EPFL, Département d'informatique, 1999.

[181] Panos M. Pardalos, H. Edwin Romeijn, and Hoang Tuy. Recent developments and trends in global optimization. *Journal of computational and Applied Mathematics*, 124(1):209–228, 2000.

[182] David Patterson and John Hennessy. *Computer Architecture: A Hardware Sofware Interface*. Morgan Kaufmann, San Franciso, California, 1996.

[183] Gregory F. Pfister. *In Search of Clusters*. Prentice Hall, 2nd edition, 1997.

[184] J. Pjesivac-Grbovíc, T. Angskun, G. Bosilca, G.E. Fagg, E. Gabriel, and J.J. Dongarra. Performance analysis of MPI collective operations. *Proc. 19th IEEE International Parallel and Distributed Processing Symposium*, 2005.

[185] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 2nd edition, 1992.

[186] Vicente Blanco Pérez. *Análisis, Prediccion y Visualización del Rendimiento de Métodos Iterativos en HPF y MPI.* PhD thesis, Universidad de Santiago de Compostela, 2002.

[187] D. Janaki Ram, T.H. Sreenivas, and K. Ganapathy Subramaniam. Parallel simulated annealing algorithms. *Journal of parallel and distributed computing*, 37(2):207–212, 1996.

[188] Daniel Reed et al. Pablo: Scalable performance tools. Technical report. http://www-pablo.cs.uiuc.edu.

[189] John J. Rehr, Fernando D. Vila, Jeffrey P. Gardner, Lucas Svec, and Micah Prange. Scientific computing in the cloud. *Computing in Science and Engineering*, 12:34–43, 2010.

[190] Inés Santé Riveira, Marcos Boullón Magán, Rafael Crecente Maseda, and David Miranda Barrós. Algorithm based on simulated annealing for land-use allocation. *Computers & Geosciences*, 34(3):259–268, 2008.

[191] Inés Santé Riveira and Rafael Crecente Maseda. RULES – sistema de ayuda para la planificación del suelo rural. *Recursos rurais: revista oficial do Instituto de Biodiversidade Agraria e Desenvolvemento Rural (IBADER)*, 1(2):25–33, 2006.

[192] Inés Santé Riveira. *Diseño de una metodología y un sistema de ayuda a la decisión espacial para la planificación de los usos del suelo rural aplicación a la comarca de Terra Chá.* PhD thesis, Universidad de Santiago de Compostela, 2005.

[193] M. Romberg. The UNICORE architecture: seamless access to distributed resources. *The Eighth International Symposium on High Performance Distributed Computing, 1999*, pages 287–293, 1999.

[194] Luis F. Romero and Emilio L. Zapata. Data distributions for sparse matrix vector multiplication. *Parallel Computing*, 21(4):583–605, 1995.

[195] Luiz De Rose, Luiz De Rose, Ying Zhang, Ying Zhang, Daniel A. Reed, and Daniel A. Reed. Svpablo: A multi-language performance analysis system. *10th International Conference on Performance Tools*, pages 352–355, 1998.

[196] Philip C. Roth and Barton P. Miller. On-line automated performance diagnosis on thousands of processes. *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming, PPoPP*, 2006.

[197] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. PWS Publishing Co., 1996.

[198] Abhijit Sahay. Hiding communication costs in bandwidth–limited parallel FFT computation. Technical Report UCB/CSD 92/722, Computer Science Division. University of Calfiornia, Berkeley, 1992.

[199] I. Santé and D. Miranda. Sistema de ayuda a la decisión espacial para la planificación de los usos del suelo. *Actas de la primera conferencia ibérica de sistemas y tecnologías de la información, Barcelos, PT*, pages 157–172, 2006.

[200] Daniel F. Savarese and Thomas Sterling. *High Performance Cluster Computing: Architectures and Systems*. Prentice Hall PTR, 1999. pp. 625–645.

[201] Peter C. Schuur. Classification of acceptance criteria for the simulated annealing algorithm. *Mathematics of Operations Research*, 22(2):266–275, 1997.

[202] Sunil K. Sharma and Brian G. Lees. A comparison of simulated annealing and GIS-based MOLA for solving the problem of multi-objective land use assessment and allocation. In *Proceedings of the 17th International Conference on Multipple Criteria Decision Analysis, Whistler, Canada*, 2004.

[203] S. Shende and A. D. Malony. The TAU parallel performance system. *International Journal of High Performance Computing Applications*, 20(2):287–331, 2006.

[204] Sameer Shende, Allen D. Malony, and Alan Morris. Optimization of instrumentation in parallel performance evaluation tools. *Proceedings of the 8th international*

conference on Applied parallel computing: state of the art in scientific computing, 2007.

[205] D.E. Singh, M. Arenaz, F.F. Rivera, J.D. Bruguera, J. Touriño, R. Doallo, M.R. Méndez, J.A. Souto, and J. Casares. Some proposals about the vector and parallel implementations of STEM-II. *Proc. 8th Int. Conf. Development and Application of Computer Techniques to Environmental Studies, ENVIROSOFT'2000, Bilbao*, pages 57–66, 2000.

[206] David B. Skillicorn, Jonathan M.D. Hill, and W.F. McColl. Questions and answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.

[207] David B. Skillicorn and Domenico Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2), 1998.

[208] D.P. Solomatine. Genetic and other global optimization algorithms – comparison and use in calibration problems. In *Hydroinformatics*, volume 98, pages 1–2, 1998.

[209] George Stantchev, Derek Juba, William Dorland, and Amitabh Varshney. Using graphics processors for high-performance computation and visualization of plasma turbulence. *Computing in Science and Engineering*, 11:52–59, 2009.

[210] Per Stenstrom. A survey of cache coherence schemes for multiprocesors. *IEEE Computer Society*, 23(6):12–24, 1990.

[211] Per Stenstrom, Truman Joe, and Anoop Gupta. Comparative performance evaluation of cache–coherent NUMA and COMA architectures. *19th International Symposium on Computer Architecture, Gold Coast, Australia*, pages 80–91, 1992.

[212] David Sundaram-Stukel and Mary K. Vernon. Predictive analysis of a wavefront application using LogGP. *7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99), Atlanta, GA, USA*, 1999.

[213] V.S. Sunderam. Pvm: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, 1990.

[214] Systat Software, Inc., San Jose California. USA. *Systat TableCurve3D, v. 5.01*, 2005.

[215] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems. Principles and Paradigms*. Prentice Hall, 2002.

[216] Valerie Taylor, Xingfu Wu, and Rick Stevens. Prophesy: An infrastructure for performance analysis and modeling of parallel and grid applications. *ACM SIGMETRICS Performance Evaluation Review*, 30(4):13–18, 2003.

[217] OASIS WSRF TC. *OASIS Web Services Resource Framework (WSRF)*, 2004. https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrf.

[218] Peng Tian, Jian Ma, and Dong-Mo Zhang. Application of the simulated annealing algorithm to the combinatorial optimisation problem with permutation property: An investigation of generation mechanism. *European Journal of Operational Research*, 118(1):81–94, 1999.

[219] J. Triantafilis, W.T. Ward, and A.B. McBratney. Land suitability assessment in the Namoi Valley of Australia, using a continuous model. *Aust. J. Soil Research*, 39(2):273–289, 2001.

[220] Ignacio Trueba, Adolfo Cazorla, José Alier, et al. Optimization of spatial allocation of agricultural activities. *Journal of agricultural engineering research*, 69(1):1–13, 1998.

[221] Joe Truman and John L. Hennessy. Evaluating the memory overhead required for COMA architectures. *Computer Architecture News (Special Issue ISCA'21 Proceedings)*, 1994.

[222] Constantino Tsallis and Daniel A. Stariolo. Generalized simulated annealing. *Physica A: Statistical Mechanics and its Applications*, 233(1):395–406, 1996.

[223] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T. Maquire, T. Sandholm, D. Snelling, and P. Vanderbilt. *Open Grid Services Infrastructure (OGSI)*, 2003. https://www.ogf.org/documents/GFD.15.pdf.

[224] M. Ujaldon, E.L. Zapata, B.M. Chapman, and H. Zima. Vienna Fortran/HPF extensions for sparse and irregular problems and their compilation. *IEEE Transactions on Parallel and Distributed Systems*, 8(10):1068–1083, 1997.

[225] L.G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[226] Rene V.V. Vidal. Applied simulated annealing. In *Applied Simulated Annealing*, volume 396 of *Lecture Notes in Economics and Mathematical Systems*. Springer Berlin Heidelberg, 1993.

[227] D. Walker. An introduction to message passing paradigms. *18th CERN School of Computing, Arles, France*, pages 165–184, 1995.

[228] R. F. Van Der Wijngaart. NAS parallel benchmarks, version 2.4. Technical Report NAS-02-007, NASA Ames Research Center, Moffett Field (USA), 2002.

[229] Kesheng Wu and Brent Milne. A survey of packages for large linear systems. Technical Report LBNL-45446, Lawrence Berkeley National Laboratory/NERSC, Berkeley, CA. USA, 2000.

[230] Xingfu Wu. *Performance Evaluation, Prediction and Visualization of Parallel Systems*. Kluwer Academic Publishers, 1999.

[231] Juekuan Yang and Yujuan Wang y Yunfei Chen. GPU accelerated molecular dynamics simulation of thermal conductivities. *Journal of Computational Physics*, 221(2):799–804, 2007.

[232] Gengbin Zheng, Gunavardhan Kakulapati, and Laxmikant V. Kale. Bigsim: A parallel simulator for performance prediction of extremely large parallel machines. *IEEE International Parallel and Distributed Processing Symposium*, 2004.

[233] Albert Y. Zomaya and Rick Kazman. Simulated annealing techniques. In *Algorithms and theory of computation handbook*, pages 33–33. Chapman & Hall/CRC, 2010.

USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA

Índice de figuras

USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA

Índice de tablas

USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA