Universitat Autònoma de Barcelona

Departament d'Arquitectura de Computadors i Sistemes Operatius

Màster en
Computació d'altes prestacions

Sintonización dinámica de aplicaciones MPI

Memoria del trabajo de investigación del "Máster en Computació d'altes prestacions", realizada por Andrea Martínez Trujillo, bajo la dirección de Dra. Anna Barbara Morajko y Dr. Joan Sorribes Gomis. Presentada en la Escuela de Ingeniería (Departamento de Arquitectura de Computadores y Sistemas Operativos)

Bellaterra, Julio 2010

Iniciación a la investigación. Trabajo de fin de máster

Máster en Computación de Altas Prestaciones.

Sintonización dinámica de aplicaciones MPI

Realizada por Andrea Martínez Trujillo en la Escuela de Ingeniería, en el Departamento de Arquitectura de Computadores y Sistemas Operativos

Dirigida por: Anna Barbara Morajko y Joan Sorribes Gomis

Firmado

Directora	Director	Estudiante

A mis padres, mis hermanas, mis cuñados, mis sobrinos y a todos mis seres queridos
por su apoyo constante y su confianza depositada en mí, y por intentar hacer que los kilómetros que nos separan no se aprecien…

A Ania, Eduardo, Joan y Tomás,
por su enseñanzas, su ayuda incondicional, por escucharme (*'amos a ver…*)…

A Lola y Emilio,
por ayudarme a disfrutar este proceso de formación investigadora…

A Claudia, Gonza, Alvaro, Moni y Ronal,
por su amistad, sus constantes ánimos, su cariño…

A Hayden,
por hacerme más fácil el día a día y contagiarme su alegría…

A María, Conchi, Mª Dolores, Sandra, Inma y Manuel,
por ser parte de mi formación y por no olvidarse de mi…

A todos mis compañeros de primer año,
por esas clases de máster repletas de risas…

A todos los miembros de CAOS,
por recibirme con los brazos abiertos…

¡Muchísimas Gracias!

Resumen

En la actualidad, la computación de altas prestaciones está siendo utilizada en multitud de campos científicos donde los distintos problemas estudiados se resuelven mediante aplicaciones paralelas/distribuidas. Estas aplicaciones requieren gran capacidad de cómputo, bien sea por la complejidad de los problemas o por la necesidad de solventar situaciones en tiempo real. Por lo tanto se debe aprovechar los recursos y altas capacidades computacionales de los sistemas paralelos en los que se ejecutan estas aplicaciones con el fin de obtener un buen rendimiento. Sin embargo, lograr este rendimiento en una aplicación ejecutándose en un sistema es una dura tarea que requiere un alto grado de experiencia, especialmente cuando se trata de aplicaciones que presentan un comportamiento dinámico o cuando se usan sistemas heterogéneos. En estos casos actualmente se plantea realizar una mejora de rendimiento automática y dinámica de las aplicaciones como mejor enfoque para el análisis del rendimiento. El presente trabajo de investigación se sitúa dentro de este ámbito de estudio y su objetivo principal es sintonizar dinámicamente mediante MATE (Monitoring, Analysis and Tuning Environment) una aplicación MPI empleada en computación de altas prestaciones que siga un paradigma Master/Worker. Las técnicas de sintonización integradas en MATE han sido desarrolladas a partir del estudio de un modelo de rendimiento que refleja los cuellos de botella propios de aplicaciones situadas bajo un paradigma Master/Worker: balanceo de carga y número de workers. La ejecución de la aplicación elegida bajo el control dinámico de MATE y de la estrategia de sintonización implementada ha permitido observar la adaptación del comportamiento de dicha aplicación a las condiciones actuales del sistema donde se ejecuta, obteniendo así una mejora de su rendimiento.

Palabras clave: análisis dinámico, sintonización dinámica, modelos de rendimiento, computación de altas prestaciones.

Resum

En l'actualitat, la computació d'altes prestacions està sent utilitzada en multitud de camps científics on els diferents problemes estudiats es resolen mitjançant aplicacions paral·leles/distribuïdes. Aquestes aplicacions requereixen gran capacitat de còmput, bé sigui per la complexitat dels problemes o per la necessitat de solucionar situacions en temps real. Per tant s'ha d'aprofitar els recursos i altes capacitats computacionals dels sistemes paral·lels en els quals s'executen aquestes aplicacions amb la finalitat d'obtenir un bon rendiment. No obstant això, assolir aquest rendiment en una aplicació executant-se en un sistema és una tasca complexa que requereix de un alt grau d'experiència, especialment quan es tracta d'aplicacions que presenten un comportament dinàmic o quan s'usen sistemes heterogenis. En aquests casos actualment es planteja realitzar una millora de rendiment automàtica i dinàmica de les aplicacions com la millor via per l'anàlisi del rendiment. El present treball d'investigació es situa dins d'aquest àmbit d'estudi i el seu objectiu principal és és sintonitzar dinàmicament mitjançant MATE (Monitoring, Analysis and Tuning Environment) una aplicació MPI empleada en computació d'altes prestacions que segueixi un paradigma Master/Worker. Les tècniques de sintonització integrades en MATE han estat desenvolupades a partir de l'estudi d'un model de rendiment que reflecteix els colls d'ampolla propis d'aplicacions situades sota un paradigma Master/Worker: balanceig de càrrega i nombre de workers. L'execució de l'aplicació triada sota el control dinàmic de MATE i de l'estratègia de sintonització implementada ha permès observar l'adaptació del comportament d'aquesta aplicació a les condicions actuals del sistema on s'executa, obtenint així una millora en el seu rendiment.

Paraules clau: anàlisi dinàmica, sintonització dinàmica, models de rendiment, computació d'altes prestacions.

Abstract

At the present time, high performance computing is used in a multitude of scientific fields, where the problems studied are resolved using parallel/distributed applications. These applications require an enormous computing capacity due to both the complexity of the problems and the necessity to solve them in real time situations. Therefore, the computational capacities and resources of the parallel systems, where these applications are executed, must be taken advantage of to attain this vital high performance. However, achieving high performance in applications executed in parallel systems is a complicated task that requires a high degree of experience, especially when dealing with applications with dynamic behaviour or those running on heterogenous systems. In these cases the use of automatic and dynamic performance improvements is proposed as a better approach to performance analysis. The research presented falls within this field of study and has the principle objective of dynamically tuning, using MATE (Monitoring, Analysis and Tuning Environment), an MPI application which employs high performance computing following the Master/Worker paradigm. The tuning techniques integrated in MATE have been developed following a study of the performance model that reflects the bottlenecks specific to the Master/Worker paradigm: load balancing and the number of workers. The execution of the chosen application under the dynamic control of MATE using the tuning strategies implemented has permitted the observation of the behaviour of said application adapting to the changing conditions in the system where it is being executed, thus obtaining an improvement in the performance.

Keywords: dynamic analysis, dynamic tuning, performance models, high performance computing.

Índice

ÍNDICE DE FIGURAS xiii

ÍNDICE DE TABLAS xv

ÍNDICE DE ECUACIONES xvii

1. INTRODUCCIÓN…… Pág.1

1.1 DESCRIPCIÓN GENERAL…….. Pág.1
1.2 OBJETIVOS.……..……. Pág.5
1.3 ORGANIZACIÓN DEL TRABAJO…..……... Pág.7

2. ANÁLISIS DE RENDIMIENTO……..……...…….. Pág.9

2.1 INTRODUCCIÓN…..……... Pág.9
2.2 ANÁLISIS CLÁSICO DE RENDIMIENTO……..... Pág.11
2.2.1 HERRAMIENTAS……...……..... Pág.12
2.2.1.1 MPICL…….. Pág.12
2.2.1.2 PARAGRAPH…… Pág.13
2.2.1.3 PABLO……. Pág.14
2.2.1.4 VAMPIR……..... Pág.15
2.3 ANÁLISIS AUTOMÁTICO DE RENDIMIENTO……..... Pág.17
2.3.1 HERRAMIENTAS……...……..... Pág.19
2.3.1.1 SCALASCA.…….. Pág.19
2.3.1.2 PERISCOPE…...……. Pág.20
2.3.1.3 TAU...…….. Pág.21
2.3.1.4 PARAVER Y DIMEMAS..……..... Pág.22
2.3.1.5 KAPPAPI…….. Pág.24
2.4 ANÁLISIS DINÁMICO DE RENDIMIENTO……..... Pág.25
2.4.1 HERRAMIENTAS……...……..... Pág.26
2.4.1.1 PARADYN...…… Pág.26
2.5 SINTONIZACIÓN DINÁMICA DE RENDIMIENTO……..... Pág.27
2.5.1 HERRAMIENTAS……...……..... Pág.29
2.5.1.1 AUTOPILOT…….. Pág.29
2.5.1.2 ACTIVE HARMONY..…… Pág.31
2.5.1.3 PERCO...…….. Pág.32
2.5.1.4 MATE...……..... Pág.34

3. MATE…….……. Pág.35

3.1 INTRODUCCIÓN…..…… Pág.35
3.2 VISIÓN GENERAL…….. Pág.36
3.2.1 SINTONIZACIÓN DINÁMICA Y AUTOMÁTICA……..... Pág.36
3.2.2 CARACTERÍSTICAS FUNCIONALES...……..... Pág.38

3.2.3 INSTRUMENTACIÓN DINÁMICA MEDIANTE DYNINST……..... Pág.39
3.3 ARQUITECTURA……... Pág.41
3.3.1 CONTROLADOR DE APLICACIÓN……..... Pág.41
3.3.2 LIBRERÍA DE MONITORIZACIÓN DINÁMICA (DMLIB)...……..... Pág.44
3.3.3 ANALIZADOR……..... Pág.45
3.4 METODOLOGÍA DE FUNCIONAMIENTO…….. Pág.46
3.5 MATE Y OTRAS HERRAMIENTAS DE SINTONIZACIÓN DINÁMICA…… Pág.48
3.6 LIMITACIONES DE MATE COMO ENTORNO DE SINTONIZACIÓN……..... Pág.49

4. MODELO DE RENDIMIENTO PARA APLICACIONES MASTER/WORKER..…….. Pág.51

4.1 INTRODUCCIÓN…….…… Pág.51
4.2 BALANCEO DE CARGA ENTRE LOS WORKERS...……...…….. Pág.53
4.2.1 DYNAMIC ADJUSTING FACTORING (DAF).……..... Pág.55
4.3 DETERMINACIÓN DEL NÚMERO DE WORKERS...……....…….. Pág.58
4.4 DEFINICIÓN DEL MODELO DE RENDIMIENTO PARA SINTONIZACIÓN DINÁMICA..... Pág.59

5. DESARROLLO DEL MODELO DE RENDIMIENTO EN MATE…...…...……. Pág.63

5.1. INTRODUCCIÓN…… Pág.63
5.2. XFIRE…….. Pág.65
5.2.1 SIMULADORES DE INCENDIOS FORESTALES……..... Pág.65
5.2.1 VISIÓN GENERAL……...……..... Pág.66
5.2.1 ADAPTACIÓN DE XFIRE AL MODELO DE RENDIMIENTO…..... Pág.68
5.3 METODOLOGÍA DE DISEÑO DEL TUNLET……..... Pág.69
5.3.1 METODOLOGÍA…...……...……..... Pág.69
5.3.1.1 INTERPRETACIÓN DEL MODELO DE RENDIMIENTO..…… Pág.71
5.3.1.2 REQUERIMIENTOS DE MATE…….. Pág.73
5.4 DESARROLLO DEL TUNLET……..... Pág.74
5.4.1 BALANCEO DE LA CARGA ENTRE LOS WORKERS……...……..... Pág.75
5.4.1.1 INTERPRETACIÓN DE LA TÉCNICA DE SINTONIZACIÓN…….. Pág.75
5.4.2 ADAPTACIÓN DEL NÚMERO DE WORKERS……..…...……..... Pág.81
5.4.2.1 INTERPRETACIÓN DE LA TÉCNICA DE SINTONIZACIÓN…….. Pág.82
5.4.3 INTEGRACIÓN DE LAS TÉCNICAS DE SINTONIZACIÓN EN EL TUNLET…….. Pág.86
5.5 RESULTADOS EXPERIMENTALES...……..... Pág.88

6. CONCLUSIONES Y TRABAJO FUTURO……. Pág.95

6.1 CONCLUSIONES……. Pág.95
6.2 TRABAJO FUTURO……...……... Pág.98

BIBLIOGRAFÍA Pág.101

Índice de figuras

Figura 2.1. Proceso cíclico de mejora del rendimiento…… Pág.10
Figura 2.2. Aproximación clásica del análisis de rendimiento…… Pág.11
Figura 2.3 Arquitectura de análisis distribuido de VampirServer…… Pág.16
Figura 2.4. Aproximación automática del análisis de rendimiento…….. Pág.18
Figura 2.5 Esquema de la arquitectura de análisis distribuida de Periscope. El sistema de análisis de rendimiento de Periscope está compuesto por un número de agentes de análisis estructurados de forma jerárquica……... Pág.21
Figura 2.6 Aproximación dinámica del análisis de rendimiento…….. Pág.26
Figura 2.7 Aproximación dinámica de la sintonización de rendimiento…….. Pág.28
Figura 2.8 Proceso de gestión en autopilot…… Pág.30
Figura 2.9 Proceso de decisión basado en lógica difusa…….. Pág.30
Figura 2.10 Sistema de sintonización automático en tiempo de ejecución de Active Harmony…… Pág.31
Figura 2.11 Arquitectura del sistema PerCo…… Pág.33
Figura 3.1 Proceso de mejora de rendimiento de MATE……..... Pág.37
Figura 3.2. Abstracción usada en Dyninst…… Pág.40
Figura 3.3. Arquitectura interna del Controlador de Aplicación…….. Pág.42
Figura 3.4 Arquitectura de MATE sintonizando dinámicamente para MPI…… Pág.47
Figura 4.1 Técnica de balanceo de carga……. Pág.54
Figura 5.1 Ciclo de operación del modelo de André-Viegas…… Pág.66
Figura 5.2 Interrelación entre la aplicación, el modelo de rendimiento y el tunlet….. Pág.69
Figura 5.3 Sobrecarga relativa (%) y absoluta (segundos) introducida por MATE (sin aplicar sintonización) en la ejecución de Xfire…….. Pág.90
Figura 5.4 Comportamiento de la aplicación Xfire durante tres iteraciones…… Pág.91
Figura 5.5 Comparativa de los tiempos de ejecución obtenidos para los distintos escenarios planteados…… Pág.92
Figura 5.6 Estructura de la línea de fuego inicial y el umbral de desbalanceo empleado en las pruebas experimentales con Xfire…….. Pág.93

Índice de tablas

Tabla 4.1 Definición de la estrategia de balanceo de carga para su uso bajo sintonización dinámica…….. Pág.60

Tabla 4.2 Definición de la estrategia de determinación del número de workers para su uso bajo sintonización dinámica……... Pág.61

Tabla 5.1 Información sobre los eventos para la técnica de sintonización de balanceo de carga…….. Pág.80

Tabla 5.2 Información sobre los eventos para la técnica de sintonización para adaptar el número de workers……... Pág.86

Tabla 5.3 Información sobre los eventos para las 2 técnicas de sintonización implementadas……... Pág.87

Tabla 5.4 Características del entorno donde se han realizado las pruebas experiementales……. Pág.89

Tabla 5.5 Tiempos de ejecución de Xfire considerando distinto número de workers en los tres escenarios de ejecución presentados (en segundos)……. Pág.89

Tabla 5.6 Tiempos de ejecución de Xfire considerando distinto número de workers con y sin desbalanceo sintético en los workers, y de Xfire bajo MATE con desbalanceo sintético (en segundos). Ganancia sobre el tiempo de ejecución de Xfire+desbalanceo sintético obtenida aplicando la sintonizción…….. Pág.93

Índice de ecuaciones

Ecuación 4.1 Límite para el valor esperado de el estadístico de orden P para cualquier distribución de media μ y desviación estándar σ ……..... Pág.55

Ecuación 4.2 Límite superior para el tiempo de procesamiento del primer batch siendo el tiempo media de ejecución $\mu_C F_0$ con una desviación estándar $\sigma_C F_0$…… Pág.56

Ecuación 4.3 Valor inverso del factor de partición empleado para generar el primer batch de la iteración a ser repartido entre los workers……..... Pág.56

Ecuación 4.4 Valor inverso del factor de partición empleado para generar los restantes batches de la iteración a ser repartidos entre los workers…..... Pág.56

Ecuación 4.5 Índice de eficiencia que relaciona tiempo de cómputo de los workers con el tiempo total de la iteración. Empleado para calcular el número de workers que maximiza el rendimiento de una aplicación……..... Pág.59

Ecuación 4.6 Índice de rendimiento que permite relacionar rendimiento con eficiencia en el uso de los recursos. Empleado para calcular el número de workers que maximiza el rendimiento de una aplicación……..... Pág.59

Ecuación 4.7 Tiempo de ejecución de una iteración completa del proceso master..... Pág.59

Capítulo 1

Introducción

1.1 Descripción general

En las últimas décadas, se han desarrollado en distintos campos científicos una gran cantidad de aplicaciones que resuelven problemas de elevada complejidad como son la determinación del genoma humano, el análisis de la estructura de las proteínas, predicción de desastres naturales, etc. Se trata de aplicaciones paralelas/distribuidas que emplean conjuntos de datos de gran tamaño y realizan sofisticados cálculos empleando las características propias de la computación de altas prestaciones.

Las aplicaciones paralelas/distribuidas deben resolver el problema considerado tan rápido como sea posible utilizando de forma eficiente los recursos disponibles en el sistema. En este contexto, el rendimiento de la aplicación se convierte en un aspecto clave. Cuando un programador desarrolla una aplicación, espera alcanzar unos ciertos índices de rendimiento, no muy alejados del rendimiento teórico esperado. Sin embargo, el desarrollo de este tipo de aplicaciones paralelas/distribuidas constituye una tarea difícil ya que no solo implica tener un conocimiento de modelos de programación paralela y librerías de comunicación, sino que también se incluyen algunos aspectos adicionales tales como descomposición de las tareas, mapping, concurrencia, escalabilidad, eficiencia, sincronismo, etc, lo cual determina el correcto comportamiento y rendimiento de estas aplicaciones [1].

Ciertamente, la programación eficiente de una aplicación, para obtener beneficio real del paralelismo y demás características que ofrece la computación de altas prestaciones, es un

gran reto que requiere un elevado grado de pericia. Además, una vez que la aplicación ha sido implementada, debe ser depurada y testeada sistemáticamente desde un punto de vista funcional para garantizar su exactitud. Seguidamente, se debe aplicar un proceso de mejora de rendimiento. Dicho proceso, mediante la modificación de los parámetros críticos de la aplicación, permite que ésta sea ajustada y adaptada para asegurar la no existencia de cuellos de botella durante la ejecución, y en consecuencia aumentar el rendimiento de la misma.

El proceso de mejora de rendimiento incluye 3 fases sucesivas [2]: monitorización, análisis y sintonización. Primeramente durante la fase de monitorización, se captura la información o medidas de rendimiento, las cuales proporcionan los datos necesarios sobre el comportamiento de la aplicación. A continuación mediante el análisis de la información recopilada, se buscan los cuellos de botella, se deducen sus causas, intentando determinar cuáles son las acciones correctas para eliminarlos. Finalmente, se aplican los cambios decididos sobre el código de la aplicación con el fin de solventar los problemas y mejorar el rendimiento. Como consecuencia, los usuarios finales están forzados a conocer muy bien la aplicación, las diferentes capas software involucradas y el comportamiento del sistema distribuido sobre el que se ejecuta la aplicación. Todos estos aspectos hacen que el proceso de mejora de rendimiento sea difícil y costoso, especialmente para usuarios no expertos, debido al alto grado de pericia requerido para aumentar significativamente el rendimiento de la aplicación.

En concreto, la tarea más compleja e importante de todo el proceso de mejora es el análisis de rendimiento, debido a que en la práctica, los cuellos de botella pueden encontrarse en diferentes niveles de abstracción y además variar a lo largo de la ejecución de la aplicación.

Los problemas que provocan pérdidas de rendimiento pueden tener diversos orígenes. Algunos proceden de las comunicaciones, provocando un bloqueo inesperado en algunas funciones de comunicación; otros surgen debido a la implementación específica de la librería de comunicación, ya que el diseño o la implementación de las capas software pueden ser genéricas y no optimizadas para un particular sistema o condiciones. Características del sistema operativo también pueden comprometer el rendimiento de una aplicación, debido a que un inapropiado tamaño del buffer de gestión de mensajes a nivel de protocolo puede interferir en los tiempos de envío de los mensajes o de las capacidades hardware subyacentes, viéndose afectada la velocidad de ejecución de dicha aplicación.

Estos ejemplos muestran la complejidad del proceso que se ha de seguir para incrementar el rendimiento de la aplicación, poniendo de manifiesto la necesidad de usar herramientas automáticas para simplificar y acelerar el proceso de sintonización del rendimiento. Afortunadamente, a lo largo de los años han surgido distintas aproximaciones y herramientas con el objetivo de ayudar al usuario en las diferentes fases de este proceso. Estas

herramientas bajo diferentes enfoques de análisis y mejoras de rendimiento, han sido diseñadas con la finalidad de hacer más cómodo el proceso de evaluación de aplicaciones bajo entornos paralelos.

Inicialmente, estas aproximaciones o herramientas se basaban en un enfoque estático, mediante el cual se visualizaba gráficamente el comportamiento de la aplicación paralela/distribuida una vez que ésta había finalizado su ejecución. Los usuarios de tales herramientas deben ser capaces de analizar las visualizaciones que proporciona la herramienta y tomar las decisiones correctas para mejorar el rendimiento de la aplicación. Para disminuir las dificultades de los desarrolladores y usuarios en este proceso, se propuso el análisis automático. Las herramientas [3] [4] que usan este tipo de análisis están basadas en el conocimiento de problemas de rendimiento conocidos. Tales herramientas son capaces de identificar cuellos de botella críticos y ayudar en el proceso de optimización proporcionando sugerencias las cuales exponen problemas de rendimiento y ofrecen a los desarrolladores posibles mejoras.

En este tipo de aproximaciones en las que se realiza un análisis de rendimiento estático y/o automático, los datos sobre los que se toma alguna decisión son producto de ejecuciones anteriores, lo cual hace que las modificaciones realizadas en el código fuente solo sean útiles cuando el comportamiento de la aplicación no depende de los datos de entrada o no varía a lo largo de la misma ejecución. Además se requieren que el usuario posea un cierto grado de conocimiento y experiencia con aplicaciones paralelas/distribuidas, ya que son necesarios determinados cambios en el código fuente para mejorar el rendimiento del programa. De modo que este tipo de herramientas solo son adecuadas para desarrolladores con experiencias más que para usuarios de la aplicación no expertos tales como biólogos, químicos, físicos u otros científicos. El usuario final puede que no tenga conocimiento suficiente sobre la aplicación paralela/distribuida.

Para abordar todos estos problemas, surgen herramientas que automática y dinámicamente realizan la tarea de optimización de aplicaciones paralelas/distribuidas, eximiendo al desarrollador y usuario no experto de las tareas relacionadas con la mejora de rendimiento. Estas herramientas toman medidas de rendimiento, identifican cuellos de botella y realizan las modificaciones oportunas para mejorar el rendimiento, todo en tiempo de ejecución. Es decir, realizan el proceso de optimización sobre la marcha, adaptando el comportamiento de la aplicación a las condiciones actuales del sistema. De este modo, el desarrollador o usuario final no se ve en la necesidad de conocer la estructura interna de la aplicación ni de pausar la ejecuciones para tomar decisiones.

Existen diferentes herramientas que implementan esta aproximación [5] [6]. La principal diferencia entre ellas reside en los métodos o tecnologías empleadas para realizar los

procesos de monitorización y sintonización, y en la representación del conocimiento empleado para realizar la fase de análisis de rendimiento de la aplicación: lógica difusa, heurísticas, históricos o modelos de rendimiento. En la sección 2.5 del capítulo 2, se realizará un profundo estudio de ellas.

El proceso de optimización automática y dinámica de aplicaciones paralelas es una labor compleja y difícil ya que existen muchos aspectos que deben ser considerados. Una herramienta real de sintonización debería tener en cuenta puntos clave para que el proceso de sintonización dinámico sea posible y efectivo, tales como saber definir la representación del conocimiento empleado para realizar el análisis de rendimiento, realizar de forma cuidadosa las modificaciones dinámicas en la aplicación de modo que su ejecución continúe de forma correcta, llevar a cabo el análisis de una aplicación sin conocimiento de sus estructuras internas o la modificación dinámica de aplicaciones cuya estructura es desconocida.

Entre los aspectos expuesto anteriormente es muy destacable la importancia del conocimiento empleado para realizar el análisis de la aplicación, ya que a partir de él se determinará el comportamiento de la aplicación y se detectará los problemas de rendimiento existentes en la misma. Existen varias aproximaciones para realizar este análisis como por ejemplo el uso de técnicas heurísticas, modelos de rendimiento, etc. En los métodos heurísticos algunos parámetros deben ser controlados y determinados de forma automática mediante una búsqueda heurística en el espacio de valores del parámetro. Por otro lado, los modelos de rendimiento ayudan a determinar el tiempo de ejecución mínimo de la aplicación mediante la predicción del rendimiento de la misma. Estos modelos pueden contener fórmulas y/o condiciones que facilitan la determinación del comportamiento óptimo. Estas formulas necesitan medidas extraídas de la ejecución de la aplicación. De modo que basándose en las medidas y aplicando la fórmula adecuada, el modelo de rendimiento puede estimar el comportamiento de la aplicación, por ejemplo el valor óptimo de un parámetro dado. Finalmente, la aplicación puede ser sintonizada, cambiando el valor del parámetro.

Además, bajo la aproximación de análisis dinámico hay que tener presente que en ocasiones no es posible aplicar sintonización dinámica a cualquier aplicación y en cualquier entorno. Como se deduce, dicha aproximación presenta un grado de complejidad muy elevado.

El presente trabajo de investigación de máster tiene como eje principal la herramienta MATE (Monitoring, Analysis and Tuning Environment) [7]. MATE, como su nombre indica, es un entorno que desarrolla la aproximación comentada anteriormente, es decir, es capaz de sintonizar automática y dinámicamente una aplicación paralela/distribuida basándose en el conocimiento generado por el uso de modelos de rendimiento. A partir de la funcionalidad que proporciona esta herramienta, una aplicación paralela en ejecución puede ser automática y

dinámicamente monitorizada, analizada y sintonizada sobre la marcha sin necesidad de re-compilar, re-enlazar o re-ejecutar, ya que las modificaciones son realizadas empleando la instrumentación dinámica mediante el uso de la librería Dyninst [8].

1.2 Objetivos

El propósito de las aplicaciones paralelas/distribuidas es resolver el problema considerado del modo más rápido posible utilizando los recursos disponibles. Por lo tanto, el rendimiento se convierte en uno de los aspectos más importantes. De este modo el empleo de herramientas como MATE son necesarias en el campo de la computación de altas prestaciones para un correcto rendimiento de las aplicaciones paralelas.

Partiendo de la funcionalidad que proporciona MATE surge el trabajo de investigación a realizar, el cual se encuentra situado dentro de la línea de investigación *Entornos para la evaluación del rendimiento y sintonización de aplicaciones*.

El objetivo general de este trabajo de investigación es sintonizar dinámicamente mediante MATE una aplicación MPI empleada en computación de altas prestaciones que siga un paradigma Master/Worker; este estudio permitirá lograr un conocimiento sobre la pauta de comportamiento de la aplicación y sobre la herramienta de sintonización.

La determinación de la estructura y el comportamiento de la aplicación es un aspecto clave para lograr una sintonización adecuada, eficiente y poco intrusiva. Poseer esta información permite determinar los puntos que influyen en el rendimiento de la aplicación y crear un modelo de rendimiento asociado a la estructura y características de eficiencia de la misma.

Por tanto en este trabajo se pretende conocer la estructura y comportamiento de aplicaciones basadas en el paradigma de programación paralela Master/Worker, con el fin de explotar en MATE un modelo de rendimiento que describa ese tipo de aplicaciones.

Para lograr este objetivo, quedan definidos los siguientes objetivos específicos:

- Estudiar la herramienta MATE. Esta herramienta de sintonización presenta una arquitectura formada por componentes que desempeñan funcionalidades diferenciadas desde un punto de vista lógico; además dichos componentes presentan entre ellos protocolos de comunicación definidos. Por tanto, el propósito de este estudio es obtener una comprensión general de la pauta de cada uno de los integrantes de MATE así como tener un primer contacto con la implementación de la herramienta.

- Modificar la implementación de MATE para que sea capaz de sintonizar aplicaciones basadas en la librería de paso de mensajes MPI desarrolladas en C/C++. Inicialmente MATE fue desarrollada para la sintonización de aplicaciones PVM. La decisión de realizar este cambio en la implementación tiene como fin incrementar la usabilidad de MATE, ya que en la actualidad la gran mayoría de las aplicaciones paralelas/distribuidas desarrolladas en ámbitos científicos emplean la librería MPI.

- Estudiar el modelo de rendimiento para aplicaciones Master/Worker expuesto en [9]. Este modelo engloba los problemas de rendimiento propios de este tipo de aplicaciones: balanceo de carga y número de workers. Para ello, partiendo de un modelo analítico, aplica una metodología que consta de dos fases: una primera fase que emplea una estrategia dinámica para el balanceo de carga y una segunda para adaptar el número de workers teniendo en cuenta las características actuales en las que se encuentra el sistema.

- Localizar una aplicación paralela/distribuida empleada en computación de altas prestaciones que sea una buena candidata para el estudio que se va a realizar y que siga un paradigma Master/Worker. Tras realizar una compleja búsqueda, se llego a la conclusión de que actualmente las aplicaciones Master/Worker no presentan un uso muy extendido en computación de altas prestaciones debido al cuello de botella que supone la comunicación establecida entre un único master y todos los workers.

  A pesar de ello, y con el fin de poder obtener los conocimientos deseados del presente trabajo de investigación, se optó por la elección de una aplicación paralela/distribuida desarrollada en el departamento de Arquitectura de Computadores y Sistemas Operativos de la Universidad Autónoma de Barcelona. Se trata de un simulador de incendios de fuegos forestales, denominado Xfire [10].

- Implementar el tunlet que contiene la especificación del modelo de rendimiento estudiado para aplicaciones Master/Worker

  Los tunlets son el núcleo de la sintonización automática y dinámica implementada por MATE, en términos de representación del conocimiento. Cada tunlet define e implementa una particular técnica de sintonización, es decir, la lógica para resolver un determinado problema de rendimiento mediante la encapsulación del conocimiento de dicho problemas basándose en puntos de medida, funciones de rendimiento y puntos/acciones de sintonización.

  El tunlet diseñado será implementado en C++ empleando MPI y posteriormente integrados en la herramienta MATE. Por tanto, el desarrollo del tunlet conlleva el

análisis del modelo de rendimiento propuesto con el fin de identificar los puntos de medida y sintonización que se deben de implementar.

- Realizar la experimentación necesaria que permita observar las posibles mejoras de rendimiento en la aplicación elegida tras la aplicación de la sintonización dinámica mediante la funcionalidad que proporciona MATE con el conocimiento del modelo de rendimiento estudiado integrado en el tunlet.

Tras la presentación de los objetivos, el resultado que se espera de este trabajo de investigación es obtener los conocimientos necesarios sobre el proceso de optimización automática y dinámica de aplicaciones paralelas/distribuidas, especialmente aquellos relacionados con las fase de análisis de rendimiento, en la que interviene los modelos de rendimiento, y la fase de sintonización dinámica. Además se pretende que esta investigación, y las conclusiones que se obtengan de ella, permitan comenzar con el estudio a largo plazo de la tesis doctoral que tiene como fin centrase en el análisis de las características de escalabilidad de MATE.

1.3 Organización del trabajo

El contenido de este trabajo de investigación se presenta dividido en los siguientes capítulos:

- *Capítulo 2: Análisis de rendimiento.*

  Se describe las aproximaciones existentes en el análisis de rendimiento de aplicaciones paralelas/distribuidas, desde el análisis clásico hasta la sintonización dinámica. Además se detallan algunas de las herramientas que conforman el estado del arte actual del análisis de rendimiento.

- *Capítulo 3. MATE.*

  Se centra en proporcionar una descripción general de MATE y detallar los principales aspectos relacionados con su arquitectura. Además, se expone los conceptos básicos sobre la instrumentación dinámica empleada por MATE en las fases de monitorización y sintonización. Finalmente se muestran las analogías y diferencias de MATE con otras herramientas existentes que realizan la misma labor que MATE basándose en otros métodos en el ámbito de la mejora dinámica de rendimiento de aplicaciones paralelas/distribuidas.

- *Capítulo 4. Modelo de rendimiento para aplicaciones Master/Worker.*

  Se presenta una descripción del modelo de rendimiento desarrollado para la sintonización dinámica de aplicaciones Master/Worker. Es un modelo de dos fases consistente en una estrategia para balancear la carga de los workers, y un modelo analítico para adaptar el número de workers de la aplicación.

- *Capítulo 5. Desarrollo del modelo de rendimiento en MATE.*

  Muestra el proceso de integración del conocimiento proporcionado por el modelo de rendimiento, expuesto en el capítulo anterior, en el proceso de sintonización dinámica y automática que implementa MATE. También se detallan las características de la aplicación Master/Worker elegida para ser sintonizada. Finalmente se exponen las pruebas experimentales y resultados obtenidos de ejecución de la aplicación elegida tras ser sintonizada bajo MATE empleando como lógica de análisis y sintonización de rendimiento el estudiado modelo de rendimiento.

- *Capítulo 6. Conclusiones y trabajo futuro*

  Resumen el trabajo de investigación realizado, extrayendo las conclusiones derivadas del análisis y estudios realizados. Además se presentan las líneas abiertas presentes en esta área de investigación a través de las cuales se pretende dirigir el trabajo futuro.

Capítulo 2

Análisis de rendimiento

2.1 Introducción

Uno de los principales propósitos de las aplicaciones paralelas y distribuidas es aprovechar los recursos y las altas capacidades computacionales de los sistemas paralelos. Por tanto obtener un buen rendimiento en dichas aplicaciones se ha convertido en un punto clave del procesamiento paralelo/distribuido. Sin embargo, lograr este rendimiento en una aplicación ejecutándose en un sistema es una dura tarea que requiere un alto grado de experiencia. De este modo, los usuarios, con el objetivo de mejorar el comportamiento de sus aplicaciones, deben enfrentarse al proceso de optimización de rendimiento. Se trata de un proceso cíclico, mostrado en la figura 2.1, compuesto principalmente por 3 fases fundamentales [2]:

- Fase de monitorización, mediante la que se añade a la aplicación cierta información de instrumentación que permita recopilar conocimiento acerca del comportamiento de la aplicación.
- Fase de análisis, durante la cual se inspecciona la información recopilada en la fase de monitorización y la información estática del programa con el objetivo de detectar problemas de rendimiento, deducir las causas y determinar soluciones.
- Fase de sintonización, en la que se subsanan los posibles errores de rendimiento presentes en el comportamiento aplicando los cambios oportunos en el código de la aplicación.

Se han desarrollado una gran cantidad de herramientas de análisis de rendimiento que ayudan al usuario a tratar los problemas de rendimiento de su aplicación. Estas pueden clasificarse en herramientas de monitorización, de análisis y/o de sintonización, aunque existen herramientas que realizan varias de estas acciones, ayudando al usuario en más de un único nivel.

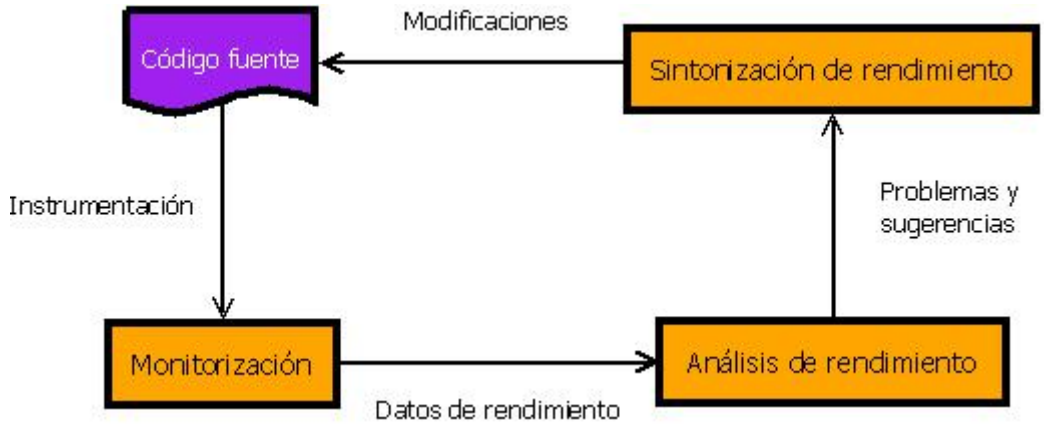

Figura 2.1. Proceso cíclico de mejora del rendimiento

Las herramientas de monitorización consta generalmente de dos partes: una librería o conjunto de librerías que permiten la inserción de instrumentación y rutinas para medir y almacenar los datos; y una seria de módulos cuya funcionalidad ofrece la posibilidad de mostrar los datos generados durante el monitoreo. Hay que tener en cuenta que la instrumentación puede afectar a las características de rendimiento de la aplicación paralela.

El objetivo de las herramientas de análisis es examinar automáticamente la información generada durante la monitorización para determinar los posibles cuellos de botella de la aplicación paralela. Para ello debe poseer conocimiento sobre los problemas de rendimiento posibles en la aplicación con el fin de proporcionar una solución de los mismos.

En lo referente a las herramientas de sintonización su intención es automatizar la inserción de las modificaciones oportunas en el código de la aplicación paralela para eliminar las imprecisiones en el rendimiento detectadas en la fase de análisis.

A lo largo de los años, con el desarrollo de la computación de altas prestaciones, se han propuesto varias aproximaciones de monitorización, análisis y sintonización de rendimiento para cooperar con el usuario en la mejora de sus aplicaciones. En las siguientes secciones se muestra un resumen de estas aproximaciones así como ejemplos de herramientas actuales que funcionalmente se sitúan dentro de las mismas.

2.2 Análisis clásico del rendimiento

La aproximación clásica de análisis de rendimiento está basada en la visualización de la ejecución de la aplicación paralela una vez que ésta ha terminado de ejecutarse. Este proceso, mostrado en la figura 2.2 recibe el nombre de análisis de rendimiento post-mortem.

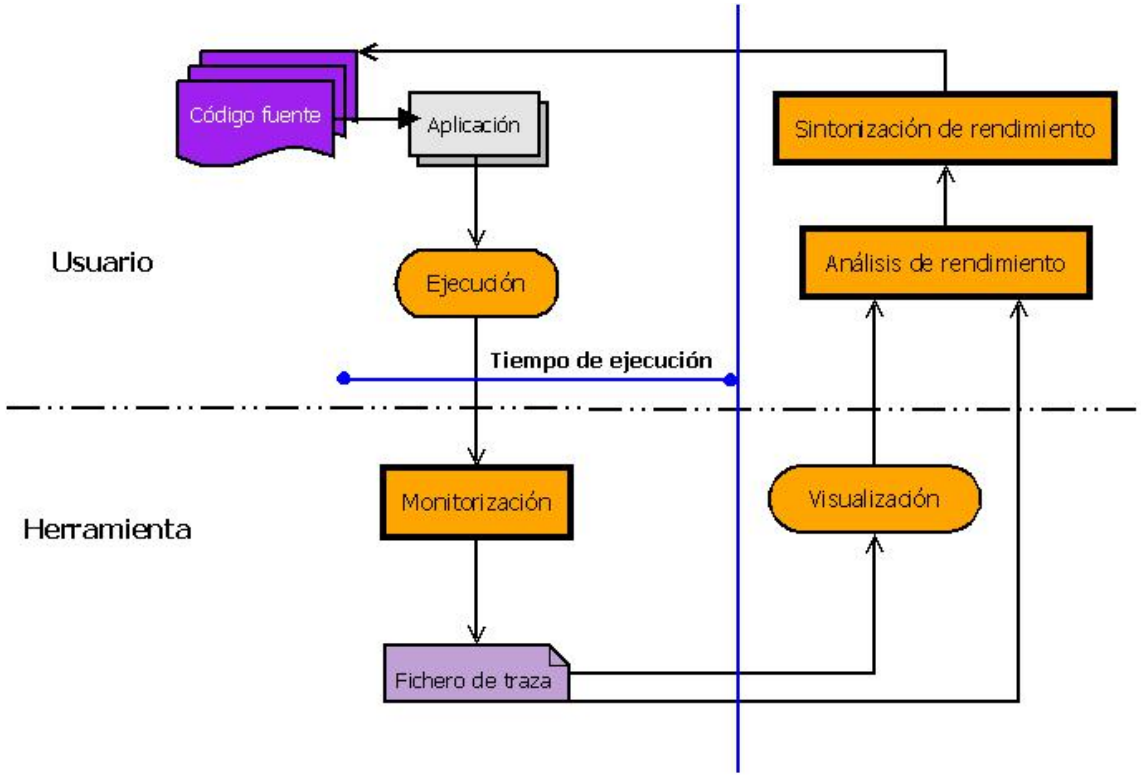

Figura 2.2. Aproximación clásica del análisis de rendimiento

Generalmente, las herramientas situadas en esta aproximación reflejan información específica sobre el comportamiento de la aplicación mediante diferentes vistas gráficas y numéricas. Para ello, primeramente se requiere el uso de herramientas que realicen la monitorización para obtener datos de rendimiento de la ejecución del programa paralelo. La inserción de la instrumentación se puede realizar de forma estática por la herramienta o bien manualmente por el usuario. El proceso de monitorización se puede realizar siguiendo varias técnicas:

- Basadas en tiempo de ejecución, mediante las que se detecta donde la aplicación paralela emplea la mayor parte del tiempo.
- Basadas en contadores, que indican el número de ocurrencias de un determinado evento en la aplicación.
- Basadas en muestreo, las cuales generan medidas periódicas sobre el estado de la aplicación.
- Basadas en trazas de eventos, que proporcionan información asociada a eventos concretos definidos en la aplicación paralela.

Con los datos almacenados en un fichero de traza de la aplicación, las herramientas de visualización generan gráficos sobre la pauta que sigue la aplicación, tales como diagramas de Gantt, diagramas circulares, de barra, etc. La información mostrada debe corresponder con aspectos relacionados con paso de mensajes, comunicaciones colectivas, ejecución de rutinas de la aplicación entre otras. Finalmente, el usuario debe analizar esas representaciones gráficas buscando problemas de rendimiento, determinando las causas de dichos problemas y cambiando el código fuente manualmente. De este modo, el proceso global se repite, volviendo a compilar, enlazar y ejecutar la aplicación, hasta que el rendimiento deseado sea alcanzado.

El análisis de rendimiento clásico requiere un elevado grado de experiencia en programación paralela para ser llevado a cabo de modo eficiente, de modo que constituye una tarea difícil especialmente para usuarios no expertos. La complejidad de esta tarea se debe principalmente a la interpretación y tamaño del fichero de traza el cual es proporcional al tamaño y el tiempo de ejecución de la aplicación. Además, esta aproximación no es fiable cuando las aplicaciones o los entornos de ejecución tienen un comportamiento dinámico. Muchas aplicaciones tienen un comportamiento diferente según los datos de entrada o incluso pueden variar durante la misma ejecución. Además muchas herramientas de visualización no escalan bien, por lo que cuando el número de procesos implicados en la aplicación es muy elevado, los gráficos generados son ilegibles.

2.2.1 Herramientas

2.2.1.1 MPICL

MPICL [11] es una librería para instrumentación y monitorización desarrollada en el Laboratorio Nacional de Oak Ridge en 1997.

Su funcionamiento se basa en la recopilación de información sobre comunicación y eventos definidos por el usuario en programas paralelos usando MPI escritos en C o FORTRAN. Para ello, emplea la interfaz de *profiling* de MPI que interceptar automáticamente las llamadas a las rutinas de comunicación de MPI, eliminado la necesidad de añadir más que unas cuantas sentencias al código fuente para recopilar información.

MPICL instrumenta el código de la aplicación, primeramente usando rutinas en C que consultan el sistema de reloj y guarda información de eventos específicos en buffers internos Esta librería puede ser usada de dos formas distintas. Puede ser usada para realizar *profiling*, resumiendo el número de ocurrencias, estadísticas, y el tiempo gastado en comunicación. También puede ser usada para recopilar trazas de eventos, las cuales pueden ser visualizadas empleando una herramienta de visualización como Paragraph [12]. Finalmente MPICL genera un fichero de trazas [13], que contiene un registro de evento por línea, y cada registro consiste

en un conjunto de valores numéricos que especifican el tipo de evento, marcas de tiempo, el número de procesador, la longitud del mensaje, y otra información similar.

MPICL es una extensión de la librería PICL [14], un paquete software que proporciona una interfaz portable de paso de mensajes antes de que apareciera el estándar MPI. Los comandos de paso de mensajes de PICL simplemente llamaban a los comandos nativos subyacentes de cada máquina en la cual estaba implementado. Un usuario MPI no necesita saber nada sobre el paso de mensajes PICL, lo cual significa que MPICL puede ser usada para recopilar datos de rendimiento para programas no implementados con la librería MPI. Pero información sobre eventos de comunicación solo se recopila si se usa MPI, los comandos de paso de mensajes de PICL, o si el usuario instrumenta la capa de paso de mensajes empleando los comandos de instrumentación de MPICL.

Una característica destacable de MPICL es que intenta minimizar el overhead introducido por la recolección de información almacenando los ficheros de traza en la memoria local de cada procesador, después los descarga al disco solo cuando la aplicación haya terminado su ejecución. Sin embargo, tal monitorización introduce un coste extra que el caso de MPICL es una cantidad fija que se añade al coste de envío de cada mensaje. De este modo, la perturbación total es función de la frecuencia y del volumen del tráfico de comunicación, lo cual varía de máquina a máquina. Esta perturbación es normalmente bastante pequeña para que el comportamiento de la aplicación no se vea afectado.

2.2.1.2 Paragraph

Paragraph [12] es una herramienta de visualización que proporciona una representación gráfica, detallada y animada así como resúmenes de rendimiento gráficos de programas paralelos que usan MPI. Fue desarrollada por la Universidad de Illinois y la Universidad de Tennessee en 1995.

Paragraph tiene una relación consumidor-productor con MPICL: Paragraph emplea exclusivamente las trazas de datos que genera MPICL. De este modo, usando MPICL junto con MPI, el usuario puede crear los ficheros de datos necesarios para usar Paragraph, para analizar el comportamiento y el rendimiento de programas paralelos.

Está escrito en C y su estructura software está compuesta por un bucle de eventos y un switch que selecciona acciones basándose en la naturaleza de cada evento. Hay dos colas de eventos separadas: una cola eventos producidos por el usuario (clicks de ratón, pulsaciones de teclas…) y una cola de trazas de eventos producidas por el programa paralelo bajo estudio. Paragraph se alterna entre estas dos colas para proporcionar una representación dinámica del programa paralelo y una respuesta interactiva con el usuario.

Aunque Paragraph solo es usado en la etapa de post-procesado, usando un fichero de traza generado durante la ejecución de un programa paralelo y almacenándolo para su posterior estudio, los datos de la visualización podrían en principio ir llegando a la estación de trabajo gráfica al mismo tiempo que la aplicación paralela se ejecuta en la máquina paralela.

2.2.1.3 Pablo

Pablo [15] es un entorno de análisis de rendimiento diseñado para desarrollar captura, análisis y presentación de datos en una gran variedad de sistemas paralelos escalables. Fue diseñado por la Universidad de Illinois en 1993.

Su infraestructura se divide en dos componentes principales:

- Un software portable para realizar la instrumentación.
- Un componente que realiza el análisis de rendimiento

El software para la instrumentación permite la especificación interactiva de puntos de instrumentación en el código fuente. Este software puede ser usado para recopilar datos de rendimiento sobre cualquier sistema o código de aplicación. Como parte de la instrumentación, se desarrollaron 3 módulos software: una interfaz gráfica para especificación de la instrumentación, analizadores en C o Fortran que emiten código fuente instrumentado y una librería de captura de eventos de rendimiento en formato estándar [16] [17] generados por el código instrumentado cuando es ejecutado en sistemas paralelos de memoria distribuida. Pablo permite 3 tipos de monitorización: *tracing*, *profiling* e intervalos de tiempo. Los eventos de traceo representa la ocurrencia de una acción específica (por ejemplo un procedimiento en concreto es llamado por un procesador en un momento determinado), de manera que cada evento produce una entrada en el fichero de datos de rendimiento. En el caso de los eventos de conteo, estos no contienen datos de usuario, solo cuenta el número de veces que tiene lugar un determinado evento o acción. La librería de captura de datos de Pablo permite cuando almacenar un registro de eventos de conteo en un fichero de datos (por ejemplo cuando el contador alcance una determinada cantidad). Finalmente los eventos de intervalos de tiempo asocian un evento con dos puntos del código fuente. Cada ocurrencia produce un evento que contiene el tiempo que ha transcurrido durante la ejecución del código fuente situado entre los dos puntos especificados.

El componente de análisis de rendimiento de Pablo consiste en un conjunto de módulos de transformación de datos que pueden ser gráficamente interconectados, para formar un grafo acíclico y dirigido de datos de análisis. Los datos de rendimiento fluyen a través de los nodos del grafo y son transformados para ofrecer las métricas de rendimiento deseadas.

2.2.1.4 Vampir

Vampir [3] [18] es una herramienta de análisis de rendimiento que permite la visualización gráfica y análisis de los cambios de estado de un programa, mensajes punto a punto, operaciones colectivas y contadores de rendimiento hardware junto con resúmenes estadísticos. Está diseñada para ser una herramienta de fácil uso, lo cual permite a los desarrolladores visualizar rápidamente el comportamiento de su aplicación en un determinado nivel de detalle.

Comenzó a desarrollarse en el Centro de Matemática Aplicada del Centro de Investigación de Jülich y el Centro de Computación de Altas Prestaciones de la Universidad Técnica de Dresden. Vampir está disponible como producto comercial desde 1996. En el pasado, fue distribuida por German Pallas GMBH, empresa que pasó a formar parte posteriormente de la compañía Intel. La cooperación con Intel terminó en 2005. Actualmente el desarrollo de Vampir continúa por parte del Centro de Servicios de Información y Computación de Altas Prestaciones (ZIH) de la Universidad Técnica de Dresden. Hoy en día, los productos Vampir se pueden obtener directamente desde la página web.

Esta herramienta ha sido probada y ampliamente usada en la comunidad de la computación de altas prestaciones durante muchos años. Un gran número de entornos de monitorización del rendimiento como TAU [19], KOJAK [20] o VampirTrace [21] generan ficheros de trazas que son interpretables por Vampir. Desafortunadamente no soporta el fichero de traza de estructura Intel, debido a razones de licencia. Desde la versión 5.0, Vampir soporta el formato Open Trace (OTF), desarrollado por ZIH. Este formato de traza está especialmente diseñado para programas masivamente paralelos. Diferentes gráficos temporales muestran las actividades y comunicaciones de la aplicación a lo largo de los ejes de tiempo, sobre los cuales el usuario puede desplazarse y hacer zoom, con el objetivo de detectar la causa real de los problemas de rendimiento. Además permite verificar la correcta paralelización y el balanceo de carga. Vampir generara gráficos estadísticos que proporcionan resultados cuantitativos sobre porciones arbitrarias temporales. La implementación está basada en el estándar X-Windows and Motif y corre en estaciones de trabajo así como en sistemas de producción paralela. Está disponible para casi todas las plataformas de 32 y 64 bit como PCs y Clusters Linux, IBM, SGI, SUN y Apple.

Actualmente hay dos versiones de Vampir. La primera, la estación de trabajo basada en la aplicación clásica con una historia de desarrollo de más de 10 años. Su último lanzamiento constituye la versión 7.1 y data de Noviembre del 2009 [22]. La segunda, la versión más escalable y distribuida llamada VampirServer [21]. Además, hay software de instrumentación y medida conocido como VampirTrace [21].

VampirServer

Es la siguiente generación de Vampir, que presenta una implementación paralela con una escalabilidad mucho mayor. La última versión desarrollada es VampirSever2.0 y data de noviembre de 2009 [22].

Basándose en la experiencia adquirida en el desarrollo de Vampir, la nueva arquitectura, mostrada en la figura 2.3 usa una aproximación distribuida consistente en un servidor de análisis paralelo, el cual se supone que se está ejecutando en un segmento de un gran entorno de producción paralela, y un cliente de visualización de los datos de rendimiento obtenidos corriendo en otras estaciones de trabajo. El servidor es un programa paralelo el cual usa métodos de comunicación estándar tales como MPI, pthreads y sockets. La compleja preparación de los datos de rendimiento es llevada a cabo por el propio servidor. El servidor consiste en un proceso máster y un número variable de procesos worker. Ambos componentes, servidor y cliente, interactúa a través de Internet por medio de un socket estándar basado en conexiones de red. Los principales objetivos de esta aproximación paralela distribuida son los siguientes:

- Mantener los datos de rendimiento cerca de la localización donde fueron generados.
- Análisis de los datos de rendimiento en paralelo para mejorar el incremento de la escalabilidad con Speedy up del orden de 10 a 100.
- Limitar los requerimientos de ancho de banda y latencia de la red a un mínimo para permitir un rápido acceso y análisis desde entornos de trabajo remoto.

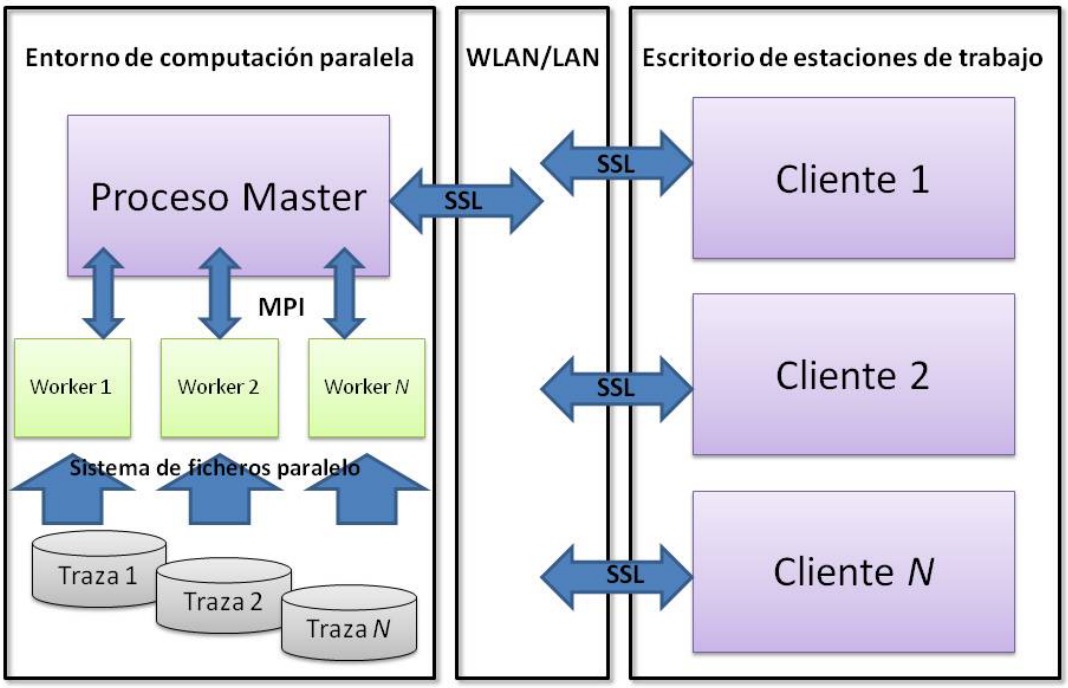

Figura 2.3 Arquitectura de análisis distribuido de VampirServer

VampirServer implementa algoritmos paralelos de análisis de eventos y gráficos personalizables que permiten un seguimiento rápido e interactivo de complejos datos procedentes de la monitorización del rendimiento. La información de la traza de eventos se almacena en memoria distribuida en la máquina de análisis paralelo. Además, grandes volúmenes de datos pueden ser analizados sin copiar grandes cantidades de datos.

VampirTrace

Proporciona una infraestructura de medida para coleccionar datos de rendimiento. Permite el desarrollo con instrumentación y facilidades para recolectar medidas en aplicaciones HPC. Cubre el análisis de aplicaciones desarrolladas con MPI y OpenMP. La instrumentación modifica la aplicación para detectar y almacenar eventos de interés generados durante la ejecución, por ejemplo una operación de comunicación MPI o una cierta llamada a función. Esto puede ser hecho a nivel de código fuente, durante la compilación o en tiempo de enlace mediante varias técnicas. La librería VampirTrace se encarga de la recogida de datos en todos los procesos. Estos datos incluyen eventos definidos por el usuario, eventos MPI, eventos OpenMP, así como información sobre temporización o localización. Además también permite obtener información mediante contadores hardware mediante PAPI. La última versión desarrollada es VampirTrace 5.8 y data de noviembre de 2009 [22].

La instrumentación automática del código fuente usando el compilador está disponible para compilador de GNU, Intel (versión 10), IBM, PGI, SUN (solo Fortran). La instrumentación binaria se desarrolla con Dynist.

Los datos de rendimiento almacenados se almacenan en un fichero usando el formato Open Trace (OTF). OTF es un rápido y eficiente formato que presenta características especiales para entrada/salida paralela. Este formato está diseñado para alcanzar un buen rendimiento en estaciones de trabajo de un único procesador así como en supercomputadores masivamente paralelos.

Su implementación está basada en el conjunto de herramientas KOJAK y es desarrollado en ZIH, en cooperación con ZAM, Centro de Investigación de Jülich, Alemania y el Laboratorio de Computación Innovadora de la Universidad de Tennesse, EEUU.

2.3 Análisis automático de rendimiento

Para reducir la complejidad que los programadores y usuarios encontraban en el análisis clásico de rendimiento, se propone un análisis automático del rendimiento de la aplicación paralela. Esta aproximación realiza un análisis automático basado en el conocimiento previo de las

características de los distintos problemas de rendimiento. La figura 2.4 muestra el ciclo de operación llevada a cabo en esta aproximación.

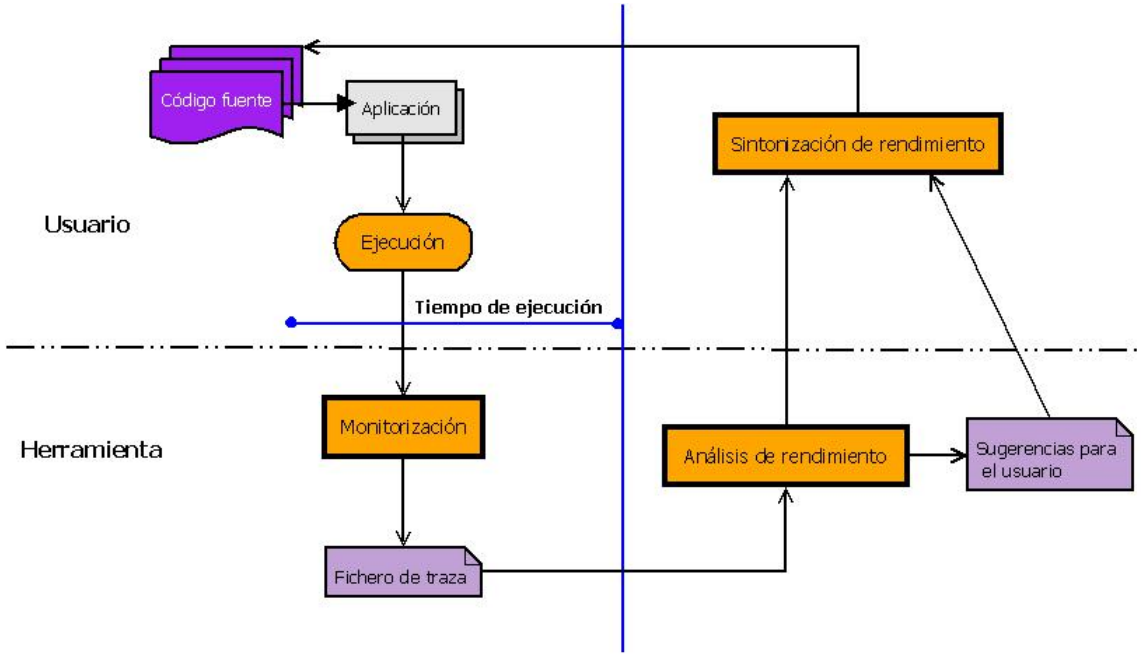

Figura 2.4. Aproximación automática del análisis de rendimiento

Las herramientas que implementan este tipo de análisis son capaces de identificar cuellos de botella críticos y ayudar a la optimización de la aplicación proporcionando sugerencias, las cuales exponen problemas de rendimiento y posibles mejoras. Para ello, la aplicación es instrumentada antes de su ejecución y la instrumentación es insertada en los puntos concretos. Una vez que los datos de rendimiento se han recopilado y almacenado en el fichero de traza, el proceso de análisis automático puede desarrollarse. Este proceso consiste en una búsqueda de problemas de rendimiento en los datos obtenidos en la ejecución, basada en información sobre posibles cuellos de botella y como encontrarlos. Cuando el proceso de análisis finaliza, el usuario puede modificar la aplicación partiendo de las sugerencias proporcionadas por el análisis, volver a compilarla y enlazarla para proceder a la siguiente ejecución.

Este tipo de análisis reduce la cantidad de tiempo que los desarrolladores invierten en análisis de rendimiento. Sin embargo de nuevo está basado en ficheros de traza que contienen una única ejecución de la aplicación, por lo que, de la misma manera que en el análisis clásico, esta aproximación no es fiable cuando las aplicaciones o los entornos de ejecución tienen un comportamiento dinámico.

2.3.1 Herramientas

2.3.1.1 Scalasca

Scalasca [23] es un conjunto de herramientas de análisis de rendimiento automático que ha sido especialmente diseñado para el uso en sistema de gran escala, incluyendo IBM Blue Gene y Cray XT, pero también está construido para su uso en plataformas HPC de pequeña y media escala. Ha sido desarrollado en el Centro de Supercomputación de Jülich, Alemania en el año 2008.

Scalasca realiza un procedimiento de análisis de rendimiento incremental que integra *profiling* en tiempo de ejecución con amplios estudios del comportamiento concurrente de la aplicación mediante trazas de eventos, adoptando una estrategia de análisis basada en sucesivas configuraciones de medidas de rendimiento. Una característica distintiva es su capacidad para detectar estados de espera que ocurren, por ejemplo, como resultado de un incorrecto balanceo de la carga. Especialmente cuando se intenta escalar aplicaciones de comunicación intensiva a grandes cantidades de procesos, tales estados de espera constituyen grandes retos para conseguir un buen rendimiento. Comparado con su antecesor, KOJAK [20], Scalasca puede detectar tales estados de espera incluso en configuraciones con grandes cantidades de procesos usando un innovador esquema paralelo de análisis de trazas.

La actual versión de Scalasca [24] permite el análisis de rendimiento de aplicaciones basadas en MPI, OpenMP, y construcciones de programación híbrida más ampliamente usadas en aplicaciones HPC altamente escalables escritas en C, C++, y Fortran en una amplia gama de plataformas actuales HPC.

En Scalasca, antes de la recopilación de cualquier dato de rendimiento, la aplicación objetivo debe ser instrumentada. Cuando se corre el código instrumentado en la máquina paralela, el usuario puede elegir entre generar un resumen con métricas de rendimiento agregadas y/o almacenar trazas de eventos individuales en tiempo de ejecución. El resumen de métricas es útil para obtener una vista general sobre el comportamiento de rendimiento. Cuando se habilita la traza, cada proceso genera un fichero de traza que contiene registros para los eventos generados de forma local.

Después de que la aplicación termine su ejecución, Scalasca carga los ficheros de traza en memoria principal y los analiza en paralelo usando tantas CPUs como han sido empleadas en la ejecución de la aplicación paralela. Durante el análisis, Scalasca genera patrones característicos indicativos de estados de espera y relaciona las propiedades de rendimiento, clasificando las instancias detectadas por categorías y cuantificando su importancia. El resultado

es un informe del análisis similar en estructura al informe resumen pero enriquecido con métricas sobre comunicación de alto nivel e ineficiencias de sincronización.

Ambos informes contienen métricas de rendimiento para cada llamada a función y recurso del sistema (proceso/hebra), pudiendo ser interactivamente explorado en una interfaz gráfica. Como alternativa a este patrón automático de búsqueda, las trazas pueden ser mezcladas y convertidas de manera que puedan visualizarse con otras herramientas como Paraver o VampirTrace, recogiendo las ventajas de sus visualizaciones desde el punto de vista temporal y su rica funcionalidad estadística.

2.3.1.2 Periscope

Periscope [25] es una herramienta escalable de análisis de rendimiento automático para aplicaciones MPI. Actualmente se encuentra bajo desarrollo en la Universidad Técnica de München en los proyectos ISAR y SILC; surge como sucesor de Peridot [26] en 2005. La última versión de Periscope data del año 2009, [27].

Consiste en un frontend, una jerarquía de comunicación, agentes de análisis y una interfaz gráfica de usuario para analizar los resultados, tal y como se muestra en la figura 2.6. Cada uno de los agentes de análisis, es decir, los nodos de la jerarquía de agentes buscan autónomamente ineficiencias en un subconjunto de procesos de la aplicación. En la parte superior de la jerarquía de agentes, el agente máster es responsable de la interacción con el usuario. El agente máster proporciona las propiedades detectadas al usuario y toma como entrada comandos que direccionan el análisis. Los agentes intermedios de la jerarquía son necesarios para la búsqueda de propiedades que no pueden ser detectadas localmente porque se deben analizar conjuntamente los datos de rendimiento de más de un nodo.

Los procesos de la aplicación están enlazados con un sistema de monitorización que proporciona la Interfaz de Peticiones de Monitorización (MRI). Los agentes se unen al monitor por medio de sockets. La MRI permite a los agentes configurar la toma de medidas para comenzar, parar, terminar la ejecución y recuperar los datos de rendimiento. El monitor actualmente solo soporta información resumida (*profiling*). La disposición de los distintos componentes de Periscope se muestra en la figura 2.5.

La aplicación y la red de agentes son comenzadas a través del proceso frontend. Éste analiza el conjunto de procesadores disponibles, determina el mapeo de la aplicación y los procesos de los agentes de análisis, y entonces lanza la ejecución de la aplicación y la jerarquía de agentes. Después de la inicialización, un comando se propaga a lo largo de la jerarquía de agentes para comenzar la búsqueda. La búsqueda se desarrolla de acuerdo a la estrategia de búsqueda seleccionada cuando comienza el proceso frontend. Al final de la búsqueda local, las

propiedades de rendimiento detectadas son enviadas a través de la jerarquía de agentes al proceso frontend. Periscope comienza su análisis desde la especificación formal de las propiedades de rendimiento. La especificación determina la condición, el valor de confianza y la severidad de las propiedades de rendimiento.

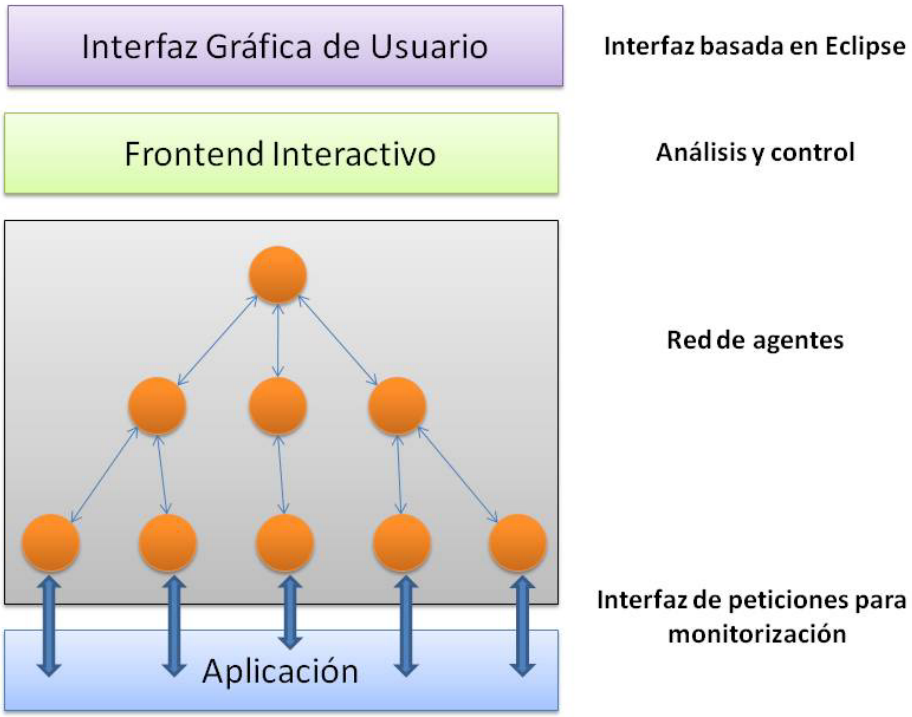

Figura 2.5 Esquema de la arquitectura de análisis distribuida de Periscope. El sistema de análisis de rendimiento de Periscope está compuesto por un número de agentes de análisis estructurados de forma jerárquica.

2.3.1.3 TAU

TAU (Tuning and Analysis Utilities) [28] [19] es un sistema de rendimiento paralelo que integra un framework y un conjunto de herramientas automáticas para instrumentación, medida, análisis y visualización del rendimiento de aplicaciones ejecutadas en sistemas paralelos de gran escala. Fue desarrollado en 1992 en la Universidad de Oregon, en EEUU, en colaboración con el Centro de Investigación de Jülich y el Laboratorio Nacional de Los Alamos. La última versión de TAU (2.19.1), fue lanzada en febrero de 2010 [29].

Una de sus principales características es el gran número de plataformas hardware y software que soporta. TAU puede ser ejecutada en la mayoría de las plataformas actuales de cómputo de altas prestaciones y permite varios lenguajes, incluyendo C, C++, Java, Python, Fortran, OpenM, MPI and Charm.

El framework está compuesto por herramientas y módulos que se integran y coordinan sus operaciones usando interfaces bien definidas y formatos de datos concretos. Su arquitectura

se organiza en tres capas (instrumentación, medida y análisis), donde en cada capa múltiples módulos están disponibles y puede ser configurados de manera flexible por el usuario.

TAU soporta un flexible modelo de instrumentación, basado en instrumentación dinámica, que permite al usuario insertar instrumentación de rendimiento llamando a la API de medidas de TAU. El concepto clave de la capa de instrumentación es que en dicha capa es donde se definen los eventos de rendimiento. El mecanismo de instrumentación de TAU permite distintos tipos de eventos que definen el rendimiento, incluyendo eventos definidos por localizaciones de código, eventos de interfaz de librerías, eventos del sistema y eventos definidos por el propio usuario. De modo que, la salida de la instrumentación es información sobre los eventos de un experimento de rendimiento. Esta información será usada por otras herramientas.

La capa de instrumentación se comunica con la capa de medida mediante la API de medida de TAU. El sistema de medida de TAU está organizado en 4 partes:

- La parte de creación y gestión de eventos determina como son procesados los eventos.
- La parte de medidas de rendimiento permite la medición de dos formas: profiling y tracing. Para cada forma TAU presenta una completa infraestructura para gestionar los datos de medida durante la ejecución a cualquier escala.
- La parte de fuentes de datos de rendimiento define que datos de rendimiento son medibles y pueden ser usados en profiling y tracing.
- La parte de sistema operativo y sistema de ejecución proporciona el acoplamiento entre el sistema de medida de TAU y el sistema paralelo subyacente. TAU especializa y optimiza su ejecución de acuerdo a las características de la plataforma disponible.

La capa de análisis y visualización permite el uso de varios módulos. Estos módulos se dividen en componentes para profiling y componentes para tracing, cuya información generada puede ser visualizada por herramientas especializadas en ellos como ParaProf o Vampir, respectivamente.

2.3.1.4 Paraver y Dimemas

Paraver [30] y Dimemas [31] son dos herramientas de análisis de rendimiento automático desarrolladas en el Centro Europeo de Paralelismo de Barcelona (CEPBA) en 1996 y 1992 respectivamente.

Paraver es una herramienta flexible para el análisis y visualización del rendimiento basada en trazas que puede ser usada para analizar cualquier información expresada en el formato de su traza de entrada de aplicaciones que empleen MPI, OpenMP, MPI+OpenMP,

Java, resúmenes de contadores hardware, actividad del sistema operativo…Además está disponible para múltiples plataformas como IRIX, AIX, Linux, Tru64.

Basada en una interfaz de usuario gráfica, Paraver fue desarrollada para responder a la necesidad de tener una percepción cualitativa global del comportamiento de las aplicaciones a través de un registro visual que permita tener una visión sobre los problemas de rendimiento presentes. Paraver proporciona una gran cantidad de información útil para mejorar las decisiones de si y dónde invertir esfuerzos en el proceso de programación con el objetivo de optimizar la aplicación.

Algunas de las principales características de Paraver son:

- Análisis cuantitativo detallado del rendimiento del programa.
- Análisis comparativo concurrente de varias trazas.
- Análisis rápido para trazas de gran tamaño.
- Permite trazas con mezcla de paso de mensajes y memoria compartida.
- Permite la personalización de la información a visualizar.
- Generación de métricas derivadas.

Paraver presenta 3 tipos de visualizaciones:

- Vista gráfica: representa el comportamiento de la aplicación en el tiempo de manera que proporcionar al usuario una comprensión general del comportamiento del programa. También permite un análisis detallado mediante el uso de patrones de identificación y relaciones de causalidad.
- Vista textual: proporciona el máximo detalle sobre la información mostrada.
- Vista de análisis: proporciona datos cuantitativos.

La visualización gráfica es suficientemente flexible para representar visualmente una gran cantidad de información y para ser la referencia para el análisis cuantitativo. Esta visualización consiste en un diagrama de tiempo con una línea para cada objeto representado. Los tipos de objetos mostrados por Paraver están muy relaciones con los conceptos de los modelos de la programación paralela (carga de trabajo, aplicación, tarea, hebra…) y con los recursos de ejecución (sistema, nodo y CPU).

Dimemas es una herramienta de análisis de rendimiento para programas basados en paso de mensajes. Permite al usuario desarrollar y sintonizar aplicaciones paralelas en una estación de trabajo, mientras proporciona una buena predicción de su rendimiento en la máquina paralela objeto de la ejecución.

El simulador Dimemas reconstruye el comportamiento temporal de la aplicación paralela en una máquina modelada por un conjunto de parámetros de rendimiento. De este modo, se pueden realizar experimentos de rendimiento de forma sencilla. El tipo de arquitecturas que se pueden simular incluyen redes de estaciones de trabajo, sistemas SMP, computadores paralelos de memoria distribuida, e incluso sistemas heterogeneos.

Dimemas soporta librerías de paso de mensajes, como PVM, MPI y PARMACS. Para la comunicación, se usa un modelo de rendimiento lineal, se tienen en cuenta además algunos efectos no lineales como conflictos en la red. Además el simulador permite especificar diferentes mapeos de tareas en los nodos.

Dimemas genera ficheros de traza válidos para dos herramientas de análisis: Paraver y Vampir.

Esta herramienta es útil en dos fases de la vida de una aplicación: durante su desarrollo, para realizar un análisis de los efectos de diferentes parámetros en el rendimiento sin requerir el uso de la arquitectura sobre la que se desea ejecutar; y después la fase de producción, para seleccionar la mejor arquitectura para ejecutar la aplicación.

Las entradas de Dimemas son: un fichero de traza y un fichero de configuración. El fichero de traza contiene los datos de una ejecución real en una máquina que captura información sobre la CPU y patrones de comunicación. Esta ejecución real puede ser hecha en cualquier tipo de máquina, incluso en máquinas uniprocesador, mapeando todos los procesos en un único procesador. Aunque el rendimiento de esa ejecución será muy bajo, el fichero de trazas de Dimemas será válido. La segunda entrada es un fichero de configuración que contienen un conjunto de parámetros que modelan la arquitectura deseada.

La salida de Dimemas puede ser simplemente un texto que contiene la predicción del tiempo empleado en la ejecución de la aplicación sobre la plataforma especificada o una visualización del fichero de trazas.

2.3.1.5 KappaPi

KappaPi (Knowledge-based Analyser of Parallel Program Applications and Performance Improver) [4] [32] es una herramienta de análisis de rendimiento automatica desarrollada en la Universidad Autónoma de Barcelona entorno a 1998.

El objetivo de esta herramienta es ayudar en la tarea del análisis de rendimiento de programas paralelos implementados bajo un paradigma de paso de mensajes (MPI o PVM), mediante la detección de los principales cuellos de botella presentes en el rendimiento, el análisis de las causan que generan estos problemas y el establecimiento de la relación entre

dichas causas y el código fuente. Para ello se basa en el análisis post-mortem de un fichero de traza y en una base de datos de conocimiento que incluye los principales cuellos de botella encontrados en aplicaciones de paso de mensajes.

En el proceso de análisis de la aplicación realizado por KappaPi se distinguen una serie de fases cuyo objetivo final es proporcionar sugerencias al usuario sobre el rendimiento actual de la aplicación:

- KappaPi obtiene los datos de ejecución de un fichero de traza.
- Análisis de eficiencia. La herramienta busca aquellos intervalos de ejecución donde la eficiencia es baja. Los intervalos se almacenan en una lista acumulada.
- Selección de las peores ineficiencias. Las ineficiencias almacenadas en la lista son ordenadas por importancia en función del tiempo acumulado y las CPUs involucradas.
- Análisis de ineficiencia. Cada una de las ineficiencias es analizada con detalle. Con la ayuda de un sistema basado en reglas, la ineficiencia es clasificada. Su motor de inferencia evalua información del programa así como información de la traza. El primer paso del proceso de deducción está basado en los eventos de la traza actual, mientras que en los siguientes pasos del análisis se requieren algunos detalles del código fuente para proporcionar sugerencias para mejorar el rendimiento.
- Sugerencias de rendimiento. Con la ayuda de la clasificación del problema, se proporciona al usuario algunos detalles de los problemas encontrados junto con algunas sugerencias para mejorar el rendimiento.

2.4 Análisis dinámico de rendimiento

El análisis de rendimiento dinámico surge con el objetivo de eliminar la necesidad de generar y almacenar enormes ficheros de traza y gestionar la cantidad de instrumentación insertada. La figura 2.6 muestra el ciclo de operación llevada a cabo en esta aproximación.

En esta aproximación el análisis de rendimiento pasa de ser post-mosterm a realizarse sobre la marcha durante la ejecución de la aplicación, de una manera completamente automática y evitando la necesidad de una instrumentación manual. Esto implica la necesidad de una monitorización constante, donde la principal ventaja es que no se necesita ficheros de traza para el análisis. Además la instrumentación puede ser dinámicamente insertada o eliminada de la aplicación mediante técnicas dinámicas de instrumentación. De este modo la fase de monitorización puede comenzar con una simple instrumentación y cuando se detectan condiciones especiales, se introduce instrumentación adicional.

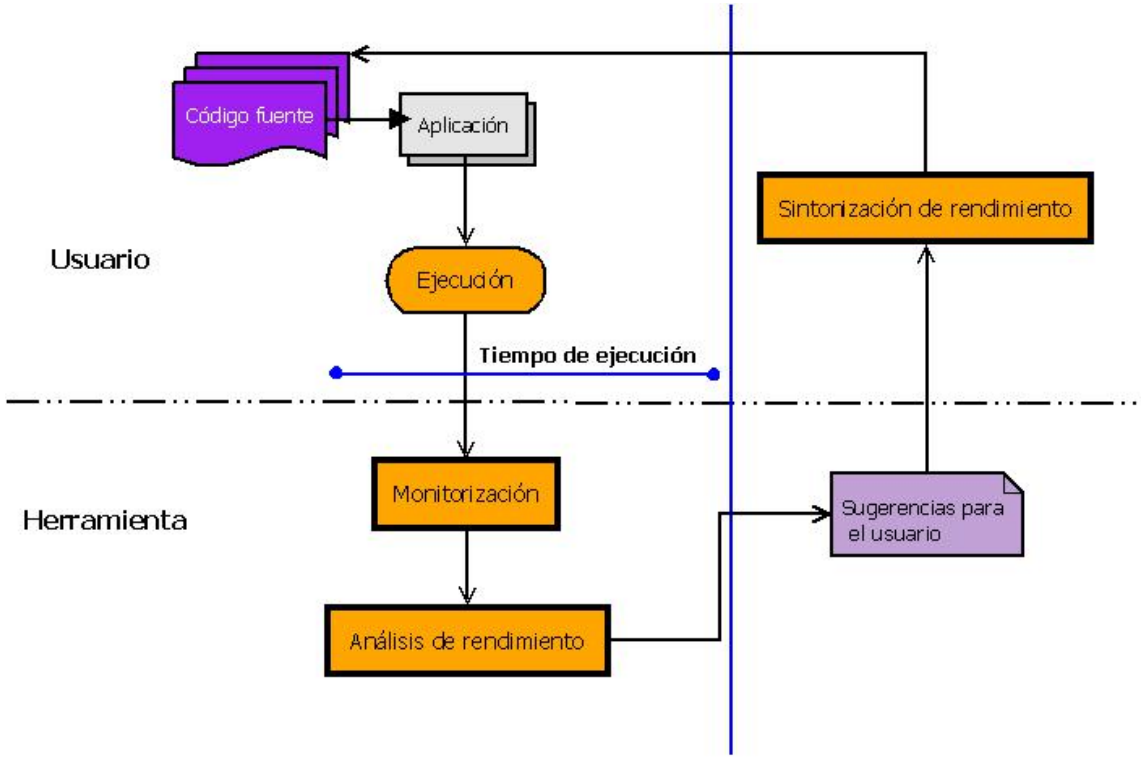

Figura 2.6 Aproximación dinámica del análisis de rendimiento

Realizar el análisis durante la ejecución de la aplicación conlleva a introducir cierto overhead dentro de ella. Por ello, el análisis debe ser relativamente simple para introducir la menor cantidad de overhead posible.

El análisis dinámico permite la detección de problemas de rendimiento de forma más rápida que las aproximaciones post-mortem. De esta manera, este análisis es adecuado para aplicaciones iterativas que presentan un amplio tiempo de ejecución con grandes volúmenes de datos. Sin embargo, requiere que el usuario pare, modifique, recompile y vuelva a ejecutar la aplicación para aplicar la sintonización. Por ello esta aproximación es adecuada para desarrolladores con experiencia más que para usuarios no expertos de la aplicación tales como químicos, biólogos, etc. Además, como en las anteriores aproximaciones, decisiones basadas en una única ejecución podrían no ser significativas cuando la aplicación presenta un comportamiento dinámico, es decir, su pauta depende de los datos de entrada o de su evolución.

2.4.1 Herramientas

2.4.1.1 Paradyn

Paradyn [33] es una herramienta de análisis desarrollada por la Universidad de Wiscosin-Madison entorno a 1994. Puede ejecutarse en la mayoría de las plataformas actuales y soporta varios lenguajes de programación como C, Fortran y permite threads y comunicación con MPI.

Los autores de Paradyn son conscientes del problema asociado al almacenamiento y análisis de grandes cantidades de datos de trazas, y se acercaron a una solución que intenta evitar estos problemas. En lugar de almacenar una traza completa de todo el comportamiento de la aplicación, Paradyn realiza un análisis de rendimiento on-line. Paradyn lleva a cabo instrumentación binaria en tiempo de ejecución [8] cuando es necesario, intentando mantener el overhead causado por la instrumentación a un nivel mínimo. De modo que el código de la instrumentación puede ser insertado o eliminado por el usuario en tiempo de ejecución. Además, también proporciona una búsqueda automática de cuellos de botella en el rendimiento denominada Performance Consultant. Además el grupo de desarrollo de Paradyn desarrolló MRNet [34], arquitectura que permite agregar datos de rendimiento de forma distribuida. Así, solo el valor final agregado se envía la herramienta de análisis de rendimiento.

Paradyn está compuesto por diferentes módulos software complementarios, todos unidos a una interfaz gráfica de usuario (GUI). Cuando los usuarios lanzan sus programas usando la GUI principal, Paradyn también lanza varios procesos demonios de monitorización en cada nodo. Cuando los usuarios seleccionan una visualización o desarrollan otra acción que requiere datos de rendimiento, la GUI se comunica con cada demonio y realiza una petición para insertar código de instrumentación en el programa en ejecución. De este modo, cada demonio realiza la instrumentación, comienza a almacenar datos, y periódicamente manda muestras de datos a la GUI principal. Estas muestras de datos son almacenadas en una base de datos round-robin, las cuales son presentadas al usuario gráficamente mediante alguna de las visualizaciones de Paradyn.

Cuando se usa el módulo de Perfomance Consultant, tiene lugar una secuencia de acciones similar a la comentada anteriormente, excepto que estas acciones son controladas por la rutina de búsqueda del módulo Perfomance Consultant en lugar de por el usuario. El proceso de búsqueda usa el modelo W^3 (Why is there a performance bottleneck? Where is it located? When did it happen?), que guía la búsqueda de ineficiencias en el rendimiento a un conjunto de cuellos de botella. Este modelo intenta responder por qué, donde y cuando la aplicación presenta un mal rendimiento relacionando las causas con especificas clases de cuellos de botella, nodos de una máquina y funciones del código fuente.

2.5 Sintonización dinámica de rendimiento

El proceso de sintonización dinámica proporciona una sintonización automática de la aplicación en tiempo de ejecución en lugar de la inserción manual de los cambios en el código fuente, desvinculando al desarrollador o usuario no experto del proceso de

sintonización de su aplicación. La figura 2.7 muestra el ciclo de operación llevada a cabo en esta aproximación.

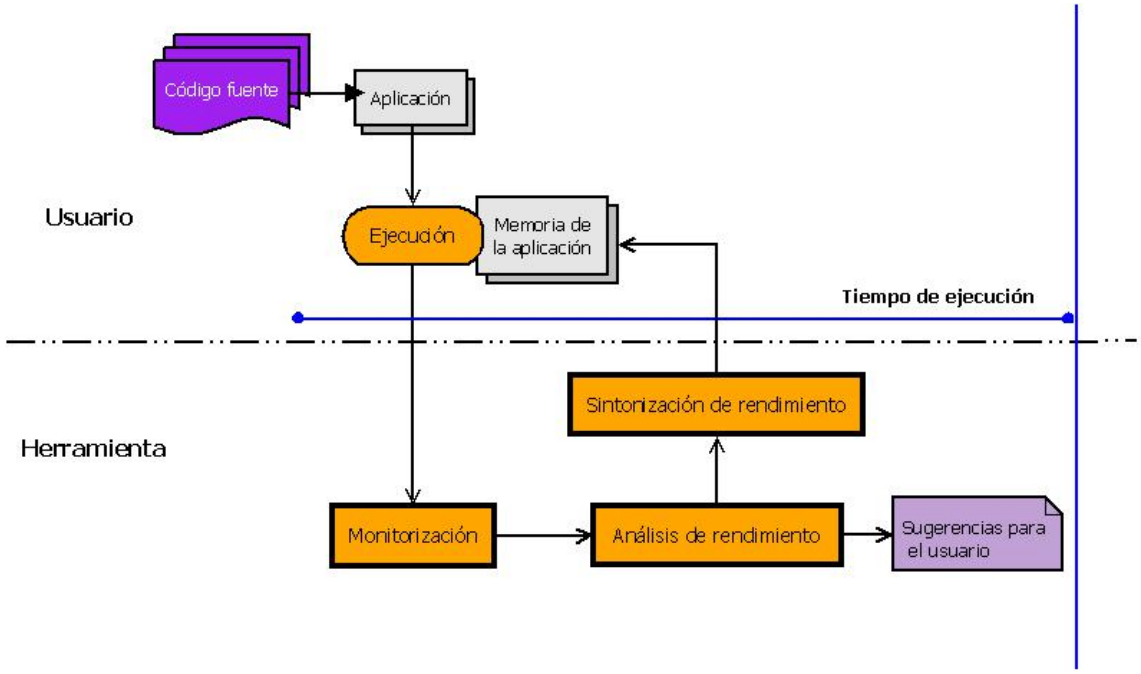

Figura 2.7 Aproximación dinámica de la sintonización de rendimiento

En esta aproximación todas las fases del proceso de optimización de rendimiento son realizadas de forma automática, dinámica y continúa durante la ejecución del programa paralelo. La aplicación es instrumentada en tiempo de ejecución de forma dinámica y automática para obtener información sobre el comportamiento de la misma. Durante la fase de análisis se busca los problemas, se detectan sus causas y se proporcionan las soluciones para eliminar esos problemas de rendimiento. Finalmente, se sintoniza la aplicación aplicando las soluciones dinámicamente. Además, mientras la aplicación está siendo sintonizada, no necesita ser compilada ni ejecutada otra vez ya que la instrumentación y las modificaciones son realizadas empleando técnicas de instrumentación dinámica.

El análisis dinámico y las modificaciones introducidas permiten la adaptación del comportamiento de la aplicación a las condiciones cambiantes de la propia aplicación o del entorno paralelo en el cual se ejecuta.

2.5.1 Herramientas

2.5.1.1 Autopilot

Autopilot [35] [6] es una infraestructura software desarrollada por la Universidad de Illinois en 1998 para la sintonización dinámica del rendimiento de entornos computacionales heterogéneos basada en bucles de control cerrados. Se basa fundamentalmente en la aplicación de técnicas de control en tiempo real para adaptar dinámicamente el sistema a las diferentes demandas y disponibilidad de recursos.

Su desarrolló se basó en la experiencia adquirida en la realización del entorno de análisis de rendimiento Pablo, propuesto por la misma Universidad.

La infraestructura de Autopilot, está formada por varios componentes software:

- Sensores y actuadores distribuidos. Los primeros capturan datos en tiempo de ejecución y los envían a los clientes; los actuadores por su parte reciben comandos desde los clientes, y ajustan el comportamiento de la aplicación y las políticas de recursos. Cada sensor y actuador está asociado con un conjunto de propiedades (nombre, tipo, dirección de red…).

  En el proceso de instrumentación los sensores y actuadores pueden operar en modos *threaded* y *no-threaded*. En el modo *threaded*, una hebra de monitorización separada se ejecuta en el mismo espacio de direcciones que la aplicación que está siendo monitorizada, y va pasivamente adquiriendo datos observando las variables compartidas y cambiando valores mediante los comandos de los actuadores. En el modo *no-threaded*, un sensor o actuador es invocado mediante una llamada a un procedimiento desde el código fuente que está siendo monitorizado.

  Para permitir la reducción de datos, todos los sensores de Autopilot presentan funciones que son invocadas cada vez que el sensor recibe datos y actúa como filtros de datos, transformando los datos originales a una forma alternativa reducida.

- Clientes, establecen comunicación directa con los sensores y actuadores. Todos los clientes remotos conectados reciben datos de los sensores, procesan estos datos, toman decisiones y envían comandos a los actuadores para implementar dichas decisiones. Además pueden cambiar el comportamiento de los sensores y actuadores (activación, tamaño del buffer…).

- Gestor Autopilot, actúa como servidor de nombres y coordina la conexión entre sensores, actuadores y clientes. El proceso de gestión se muestra en la figura 2.8. A

través de él los clientes realizan peticiones a los sensores y actuadores. Para ello, inicialmente los sensores y actuadores registran sus propiedades, con el objetivo de que cuando el cliente realiza una petición con unas características concretas, el Gestor proporciona aquellos sensores y actuadores que satisfacen las mismas.

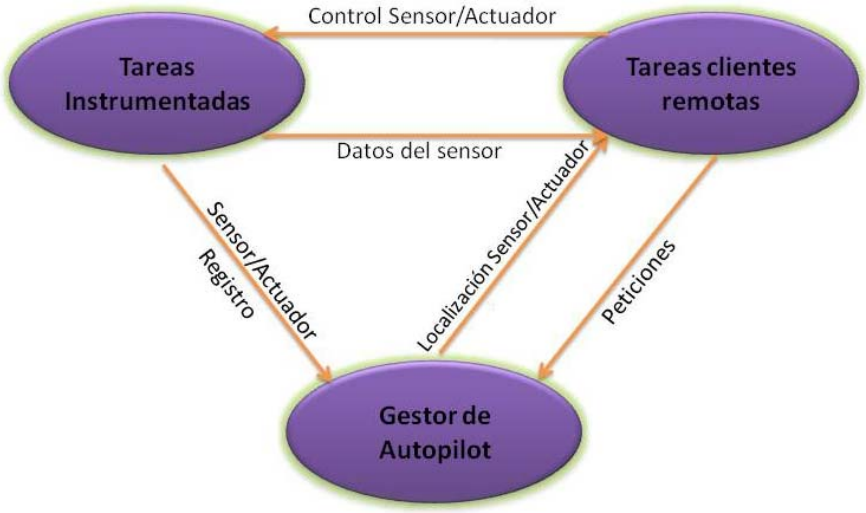

Figura 2.8 Proceso de gestión en autopilot

- Mecanismo de decisión, selecciona la política de gestión de recursos correcta basándose en las peticiones de la aplicación y en los datos de los sensores. Se estructura según un bucle de control adaptativo cerrado basado en un motor de lógica difusa, según se refleja en la figura 2.9. Esta motor toma las entradas de los sensores, fuzzifica los valores, computa la confianza relativa de cada regla, y defuzzifica los consecuentes para activar los actuadores remotos.

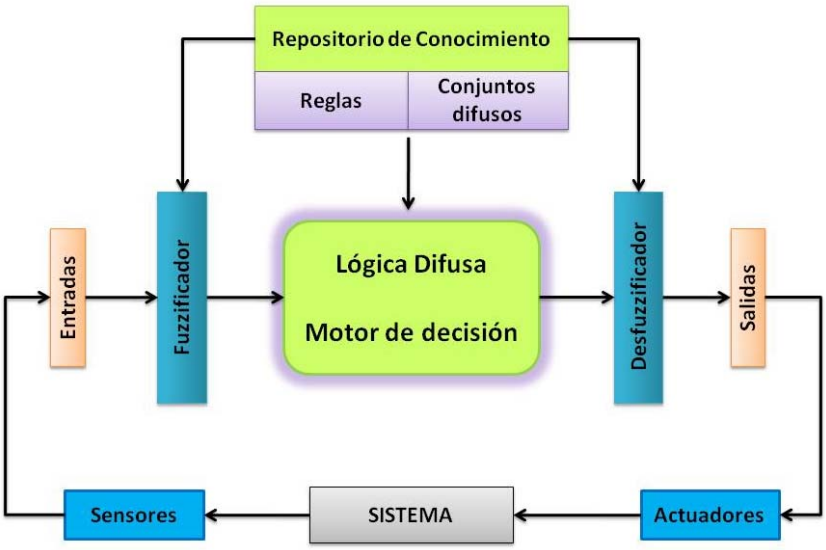

Figura 2.9 Proceso de decisión basado en lógica difusa

La comunicación entre los distintos componentes de Autopilot está construida sobre la herramienta Nexus.

2.5.1.2 Active harmony

Active Harmony [5] es un framework implementado en la Universidad de Maryland en 2002, que permite la adecuación dinámica de una aplicación a la red y a los recursos disponibles, mediante la adaptación automática de algoritmos, distribución de datos y balanceo de carga.

Su estructura está basada en un modelo cliente-servidor. El cliente es la aplicación "armonizada", la cual envía la información de rendimiento al servidor. El servidor realiza la sintonización de la aplicación y adapta las decisiones basándose en la información obtenida del cliente.

Su sistema, cuyo esquema se muestra en la figura 2.10, consiste en 3 componentes principales:

- Una API implementada en C++ que permite la integración de las librerías de la aplicación del usuario con diferentes librerías que presenta la misma o similar funcionalidad.
- El Controlador Harmony, el cual constituye la parte principal de la infraestructura del servidor Harmony.
- Algoritmos parametrizables de sintonización y optimización.

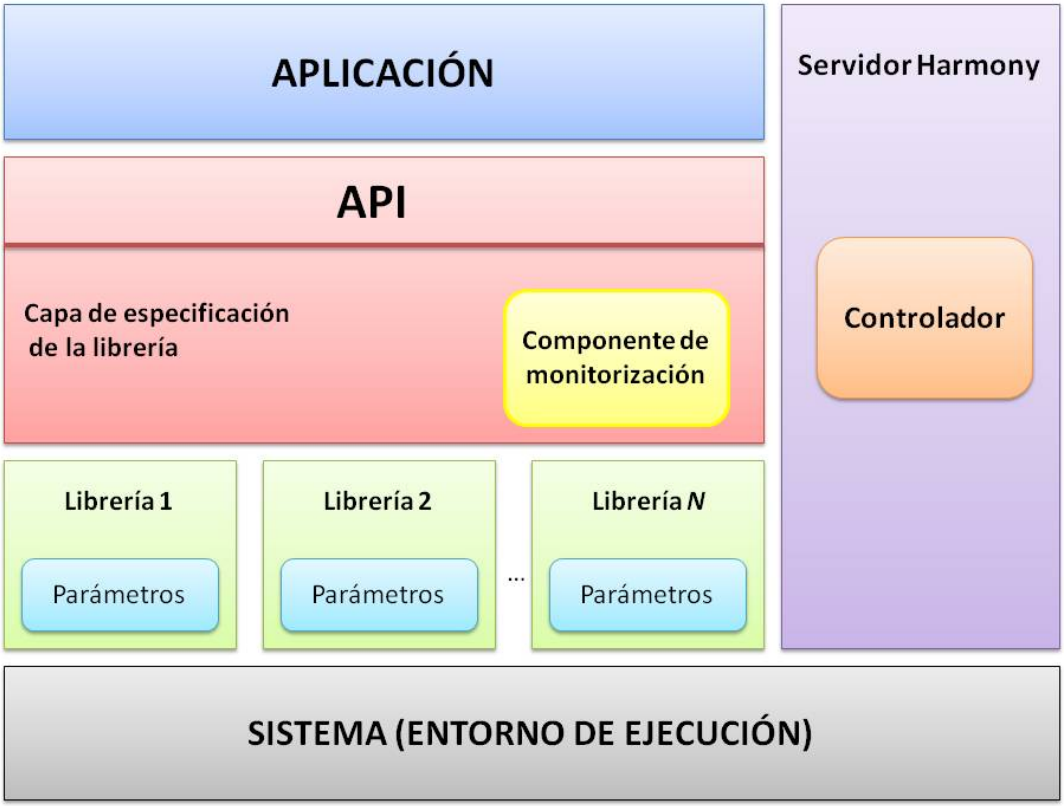

Figura 2.10 Sistema de sintonización automático en tiempo de ejecución de Active Harmony

La API permite la “armonización” de la aplicación, mediante el uso del Controlador Harmony. Su principal objetivo es ayudar a la aplicación a usar el algoritmo subyacente más apropiado. Para conseguir tal fin, primeramente se caracteriza el patrón de la aplicación y se monitoriza el rendimiento de las distintas implementaciones de librerías subyacentes. Basándose en la información recopilada, se redireccionan las llamadas a las funciones del programa de la librería subyacente seleccionada. Esta selección es la realizada a través del Controlador.

Durante la ejecución el Controlador Harmony recibe las características de la petición desde la capa formada por la API. Tras esto, el Controlador gestiona los valores de los diferentes parámetros sintonizables proporcionados por la aplicación y devuelve el algoritmo subyacente sugerido para usar de acuerdo a los resultados obtenidos en su proceso de decisión. En la implementación inicial, cuando el Controlador selecciona un determinado algoritmo, intenta explorar todos los posibles algoritmos al menos durante un breve periodo de tiempo. Para ello emplea los algoritmos basados en técnicas heurísticas mediante los cuales explora el espacio de optimización de la aplicación y ajusta los valores en la sintonización basándose en el rendimiento observado. Las métricas de rendimiento comúnmente utilizadas son el uso de los recursos por parte de la librería tales como tiempo de CPU o espacio de memoria. De modo que el Controlador intenta minimizar el valor de estas métricas de rendimiento cuando realiza la búsqueda de la librería subyacente apropiada.

Las últimas investigaciones sobre esta herramienta se basan en el estudio de la influencia de las técnicas heurísticas exploratorias del espacio para optimizar la aplicación bajo análisis [36].

2.5.1.3 PerCo

PerCo [37] es una framework para el control del rendimiento en entornos heterogéneos. Es capaz de gestionar la ejecución distribuida de aplicaciones usando migraciones, por ejemplo, en respuesta a cambios en el entorno de ejecución. PerCo monitoriza los tiempos de ejecución y reacciona de forma acorde a una estrategia de control para adaptar el rendimiento cuando tienen lugar cambios importantes en el rendimiento.

Comenzó a ser desarrollada en la Universidad de Manchester en el año 2005. Su uso está orientado para dos tipos de aplicaciones empleadas en HPC: modelos de simulación científica [38] y búsqueda distribuida en control estadístico.

PerCo está diseñado para ser una aplicación ligera con el fin de controlar el rendimiento de un programa individual en un conjunto de recursos que han sido asignados por algún gestor de recursos externo.

La aplicación a monitorizar se estructura en una serie de componentes individuales controlados cada uno de ellos por un *loader PerCo*. El conjunto de *loaders* constituyen la infraestructura de reimplementación. Cada *loader* es responsable de lanzar y mover su componente. Cada componente tiene dos interfaces. Una interfaz con otros componentes que permite la implementación de funciones que intercambian datos mediante comunicaciones entre componentes. Una interfaz de control de rendimiento para la comunicación con el componente que dirige el control del rendimiento (CPS). Esta interfaz es usada para intercambiar información de rendimiento y comandos de rendimiento. Un CPS es responsable del control local de su componente asociado. La entidad que tiene el control sobre todo el conjunto de la aplicación es el director del rendimiento de la aplicación (APS). El APS recibe datos de rendimiento desde los CPSs y contiene un repositorio de información que almacena datos históricos de rendimiento. El APS puede invocar a un predictor de rendimiento para determinar configuraciones de componentes mejoradas. La figura 2.11 muestra los distintos módulos que componen la funcionalidad de PerCo.

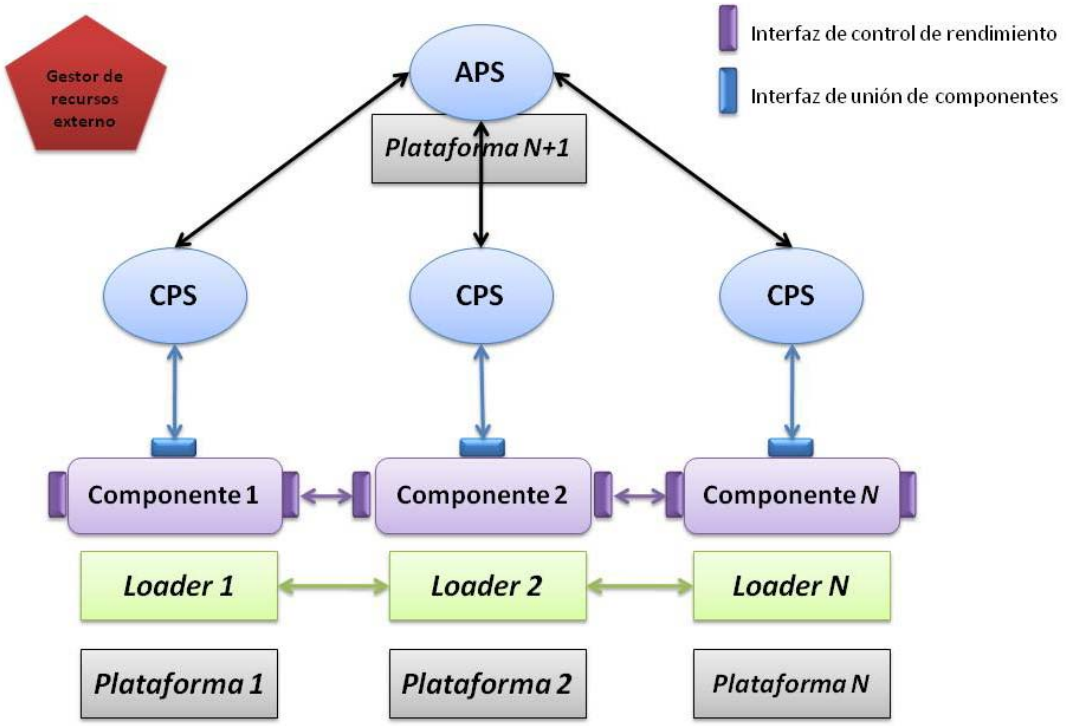

Figura 2.11 Arquitectura del sistema PerCo

Este sistema se adapta a los cambios en el rendimiento de la aplicación desarrollando políticas de balanceo de carga y tolerancia a fallos [39]. La interacción del usuario no se requiere en el proceso ya que las decisiones de reimplementación no se basan en conocimiento humano experto, si no en una política que ha sido construida en el APS. La política se basa en predicciones de rendimiento. El modelo de predicciones combina series de tiempo y técnicas de ajuste de datos para predecir el tiempo de ejecución. Las series de tiempo son usadas para

predecir el rendimiento del siguiente paso de tiempo dada la actual implementación. Las técnicas de ajuste de datos se usan para predecir el rendimiento del siguiente paso de tiempo dada una nueva implementación. En tiempo de ejecución se emplean las dos técnicas para producir dos predicciones. La predicción de mayor calidad es la que se usa. Sin embargo, la diferencia entre las dos predicciones proporciona una estimación de la calidad de la predicción.

2.5.1.4 MATE

MATE [40] (*Monitoring, Analysis and Tuning Environment*) es una herramienta que implementa una sintonización automática y dinámica de aplicaciones paralelas. Su objetivo es mejorar el rendimiento de una aplicación paralela en tiempo de ejecución, adaptándola a las condiciones variables del sistema sobre el que se ejecuta. MATE constituye el eje en el cual se basa el presente trabajo de investigación, de modo que en el capítulo 3 se detalla dicha herramienta desde un punto de vista conceptual y funcional.

Capítulo 3

MATE

3.1 Introducción

Actualmente, las aplicaciones informáticas son usadas para resolver complejos problemas en distintos ámbitos científicos como ciencia e ingeniería. Muchos de estos problemas necesitan una alta potencia de cálculo que sólo puede ser abordada por medio del procesamiento paralelo/distribuido, el cual permita aprovechar la potencia de distintos tipos de arquitecturas hardware en las que se dispone de más de un procesador. Por lo tanto, el rendimiento se convierte en uno de los aspectos más importantes en el procesamiento paralelo/distribuido.

Conseguir y mantener un buen rendimiento en aplicaciones paralelas/distribuidas es una tarea compleja, más aún cuando dichas aplicaciones o los entornos de ejecución tienen un comportamiento dinámico. Muchas aplicaciones tienen un comportamiento diferente según los datos de entrada o incluso pueden variar durante la misma ejecución. En tales casos, no merece la pena realizar un análisis de rendimiento y sintonización postmortem, ya que las conclusiones basadas en una ejecución podrían ser erróneas para otra. En estos casos actualmente se plantea realizar una sintonización dinámica y automática de la aplicación durante su ejecución sin pararla, recompilarla o reejecutarla.

Bajo este propósito se desarrolló la herramienta MATE. MATE (*Monitoring, Analysis and Tuning Environment*) [41] [42] proporciona una sintonización dinámica y automática de aplicaciones paralelas/distribuidas. Fue diseñada y desarrollada en el grupo de *Entornos para la evaluación de rendimiento y sintonización de aplicaciones* dentro del *Departamento de*

Arquitectura de Computadores y Sistemas Operativos de la Universidad Autónoma de Barcelona. Inicialmente fue creada para sintonizar aplicaciones PVM paralelas/distribuidas desarrolladas en C/C++ ejecutándose en plataformas UNIX y actualmente también está siendo desarrollada para sintonizar aplicaciones basadas en la librería de paso de mensajes MPI. Hace unos años también se desarrolló una versión orientada para entornos Grid, denominada GMATE [43].

La sintonización dinámica implementada por MATE, en concreto el uso de modelos de rendimiento en su fase de análisis, es el núcleo de este trabajo. De tal modo, en las siguientes secciones, se describen en mayor detalle las principales características, funcionalidad y arquitectura de MATE [7] [40].

3.2 Visión general

En el capítulo 2, se mostraron las diferentes aproximaciones sobre análisis de rendimiento. Como se comentó, MATE implementa una sintonización dinámica y automática del rendimiento. En la presente sección se exponen las principales consideraciones y características de MATE que hace de él un sistema de sintonización en tiempo de ejecución, útil y eficiente.

3.2.1 Sintonización dinámica y automática

El principal objetivo de MATE es mejorar el rendimiento de una aplicación paralela, adaptándola a las condiciones variables del sistema sobre el que se ejecuta. Su potencia radica en dos características principales:

- ***Sintonización dinámica***, es útil especialmente cuando las aplicaciones son ejecutadas en entornos heterogéneos o sistemas de tiempo compartido, porque las decisiones para ajustar el comportamiento de una determinada aplicación se realiza sobre la marcha, teniendo en cuenta el estado actual del sistema.
- ***Sintonización automática***, es útil porque los usuarios no deben preocuparse o participar en el proceso de búsqueda de problemas de rendimiento o en la introducción de modificaciones en la aplicación para mejorar su rendimiento.

Las decisiones de cómo mejorar el rendimiento de la aplicación se realizan mediante el conocimiento de los posibles problemas de la aplicación. Este conocimiento debe ser proporcionado por el usuario, indicando qué medidas que determinan el comportamiento de la aplicación deberían ser monitorizadas (*puntos de* medida), cómo detectar y resolver posibles problemas de rendimiento (*funciones de rendimiento*) y qué parámetros críticos en la aplicación son necesarios modificar para mejorar el rendimiento (*puntos de sintonización*). Dicho

conocimiento define un modelo de rendimiento, el cual será integrado en MATE mediante su codificación en un componente de software llamado *tunlet*.

Cuando MATE se ejecuta, carga un conjunto de tunlets los cuales proporcionan el conocimiento para mejorar y adaptar la aplicación. De este modo, un tunlet representa un modelo de rendimiento y su información es usada a lo largo del proceso de mejora de rendimiento para dirigir las fases de monitorización, análisis y sintonización. Cada tunlet es una librería compartida escrita en C/C++ que debe ser implementada usando la API de sintonización dinámica proporcionada por MATE (DTAPI).

De este modo, tal y como se muestra en el esquema de la figura 3.1, mediante la monitorización dinámica de la ejecución de la aplicación, la instrumentación se inserta de acuerdo al modelo de rendimiento definido de manera automática en la aplicación recopilando información sobre el comportamiento de la aplicación. El análisis de la información recopilada se hace evaluando las fórmulas analíticas del modelo y las soluciones son automáticamente insertadas en la aplicación, y la aplicación no necesita ser recompilada, reenlazada o reejecutada. Para modificar la aplicación en tiempo de ejecución, MATE usa la técnica llamada *instrumentación dinámica* [8].

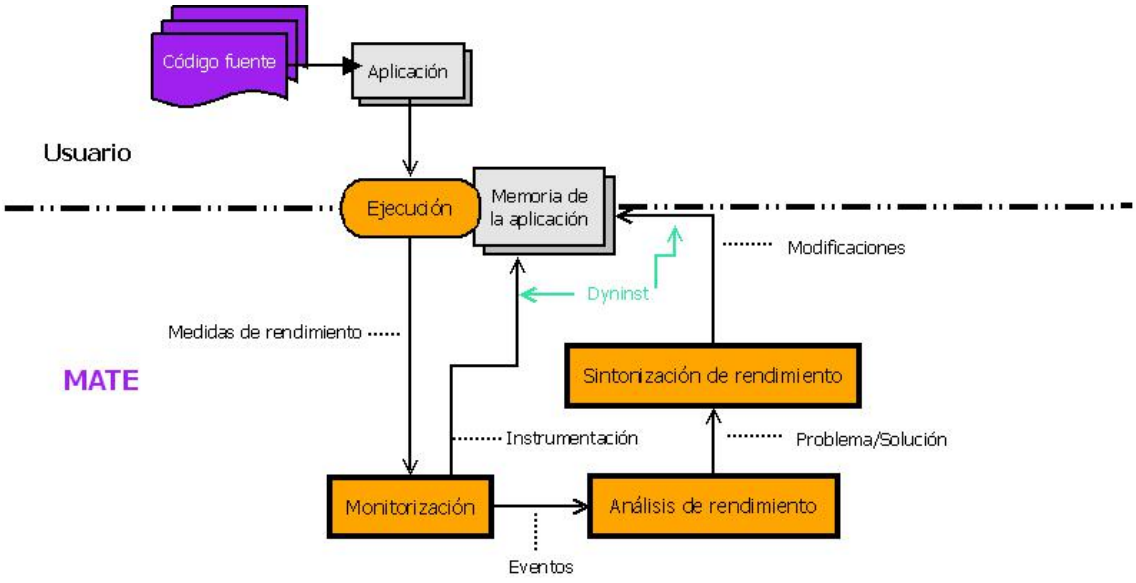

Figura 3.1 Proceso de mejora de rendimiento de MATE

La realización de estas 3 fases de forma automática hacen más fácil las tareas del usuario si se considera su intervención en el proceso de sintonización. La aproximación de la sintonización dinámica que MATE desarrolla libera al usuario de:

- Instrumentar la aplicación a mano o semiautomáticamente.
- Hacer un seguimiento de la traza de ejecución de la aplicación.

- Analizar analítica o automáticamente el rendimiento.
- Modificar y recompilar el código fuente de la aplicación.

3.2.2 Características funcionales

Desde un punto de vista funcional, en MATE se distinguen tres fases básicas y continuas que conforman el proceso de mejora de rendimiento: monitorización, análisis y modificaciones. Como se comentó anteriormente, todas estas fases se realizan continúa, automática y dinámicamente mientras el programa está en ejecución.

Para realizar correctamente el proceso de mejora de rendimiento, MATE presenta una serie de características añadidas que permiten el funcionamiento de la herramienta como un todo integrado. Estas características son:

- **Control paralelo de la aplicación.**

  El proceso de mejora de rendimiento debe actuar sobre todas las tareas ejecutadas en las distintas máquinas que conforman la aplicación con el objetivo de poder gestionar o controlar la aplicación completa.

- **Análisis global.**

  El comportamiento de la aplicación debe ser evaluado de forma global, de forma que la información recopilada de las distintas tareas que componen la aplicación debe estar centralizada para poder realizar un análisis de rendimiento global.

- **Conocimiento de la aplicación.**

  La sintonización dinámica para que sea útil y eficiente precisa, como se comentó anteriormente, que el proceso de análisis sea simple para poder tomar decisiones en un corto periodo de tiempo y que las modificaciones que se realicen en la aplicación sean claras y concisas. De tal manera, no poseer un determinado conocimiento de la aplicación, puede hacer que el proceso de sintonización pierda efectividad.

  Por tanto, es necesario proporcionar información sobre qué debería ser medido (*puntos de medida*), cómo detectar y resolver posibles problemas de rendimiento (*funciones de* rendimiento) y qué es necesario modificar para ello (*puntos de sintonización*). Así, para el correcto funcionamiento de MATE, se precisa no sólo la cooperación del usuario para definir como analizar el comportamiento de la aplicación, sino que el usuario debe de conocer también los detalles de implementación de MATE para poder desarrollar o implementar las soluciones a los posibles problemas de rendimiento de su aplicación.

- **Baja intrusión.**

  El *overhead* que causa el proceso de mejora de rendimiento debe ser mínimo para evitar afectar al rendimiento de la aplicación, ya que ambos son ejecutados concurrentemente.

- **Solventar los cuellos de botella.**

  Para solucionar los posibles problemas de rendimiento, las distintas fases del proceso de sintonización necesitan un periodo de tiempo para determinar una solución. Sin embargo, puede ocurrir que una vez que se haya aplicado la solución para un el cuello de botella, éste haya desaparecido. En este caso, la sintonización dinámica es especialmente recomendable para aquellos problemas que presenta una cierta persistencia a lo largo del tiempo.

3.2.3 Instrumentación dinámica mediante Dyninst

El principio de la instrumentación dinámica consiste en postponer la instrumentación de la aplicación hasta que ésta esté siendo ejecutada e insertar, alterar o eliminar estas modificaciones en tiempo de ejecución. Esta aproximación fue inicialmente usada en la herramienta Paradyn descrita en la sección 2.4.1.1 del capítulo 2. De modo que, el grupo de Paradyn como resultado de su investigación desarrolló una librería que permitía la instrumentación dinámica; esta librería recibe el nombre de Dyninst [44].

Dyninst es una API (*Application Program Interface*) que genera código en tiempo de ejecución y está dirigida a aplicaciones escritas en los lenguajes C/C++. La API está basada en una tecnología orientada a objetos, y proporciona un conjunto de clases y métodos que permiten al usuario la realización de una serie de acciones:

- Modificar un proceso en ejecución o comenzar un nuevo proceso.
- Crear un nuevo fragmento de código.
- Acceder y usar código y estructuras de datos existentes.
- Insertar código creado en el proceso en ejecución.
- Eliminar código previamente insertando en el programa en ejecución.

El código insertado a través de Dyninst en una aplicación, será ejecutado cuando el programa ejecute la sección de código modificada. Para que esto ocurra, la aplicación no necesita ser recompilada, ni reenlazada ni reejecutada y además Dyninst no necesita acceder al código fuente de la aplicación, ya que todo el proceso de instrumentación lo realiza gestionando la imagen del espacio de direcciones del proceso. El único requisito precisado por Dyninst es la necesidad de información de depuración sobre el programa instrumentado para ser capaz de

localizar los procedimientos y variables necesarias, de tal modo que éste debe ser compilado con la correspondiente opción habilitada.

Esta librería es usada por MATE para lograr de manera dinámica y automática dos de los procesos principales de la aproximación de mejora de rendimiento que dicha herramienta desarrolla:

- La fase de monitorización dinámica, de tal modo que mediante la instrumentación dinámica se puede añadir o eliminar código en el programa para recopilar información sobre el comportamiento de la aplicación.
- La fase de sintonización dinámica, en la cual el código de la aplicación es cambiado para mejorar su rendimiento.

Abstracciones

La librería Dyninst está basada en las siguientes abstracciones:

- Mutatee o aplicación, es el programa que va a ser instrumentado.
- Mutator, es el programa que controla y modifica la aplicación mediante Dyninst.
- Punto, es un especifico punto de la aplicación donde algún nuevo fragmento de código puede ser insertado.
- Snippet, es una representación de un fragmento de código ejecutable, el cual puede ser insertado en el programa en un punto determinado.
- Proceso, corresponde a la ejecución de una proceso.
- Imagen, constituye la representación estática del programa en disco. Cada hebra está unívocamente asociada a una imagen.

Las abstracciones usadas por Dyninst y sus interacciones se muestran en la figura 3.2.

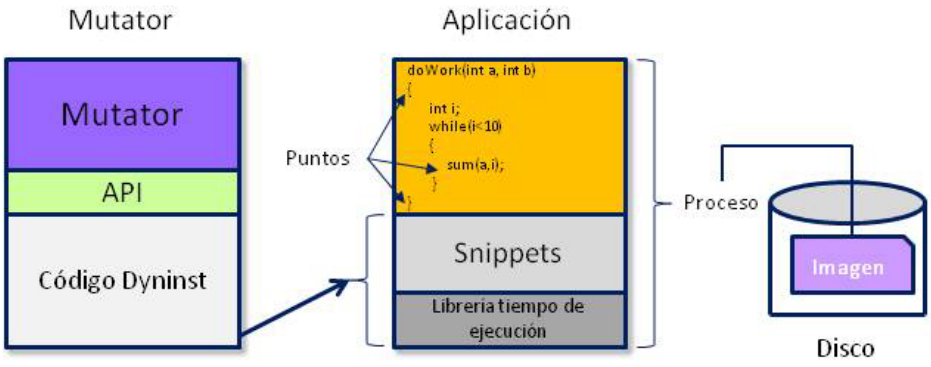

Figura 3.2. Abstracción usada en Dyninst

Para poder emplear Dyninst en el proceso de sintonización dinámica, MATE actúa como *mutator* y la aplicación a sintonizar actúa como *mutatee*. Los *snippets* y los *puntos* dependen de la información necesaria para evaluar el comportamiento de la aplicación.

3.3 Arquitectura

MATE, está compuesto por varios módulos cooperativos, que controlan e intentan mejorar el rendimiento en la ejecución de la aplicación. Los principales componentes son los siguientes:

- **Controlador de Aplicación (AC).**

  Es un proceso que controla la ejecución de la aplicación MPI. Su labor principal es gestionar los cambios que dinámicamente se realizan en las tareas individuales que componen la aplicación. Para ello se encuentra enlazado con la librería de instrumentación dinámica Dyninst, y emplea su API para generar la instrumentación y modificaciones propias de la sintonización. Como se detalló en la sección X, el código generado e insertado recibe el nombre de *snippet*. De este modo, en tiempo de ejecución, el controlador de aplicación inserta o elimina los correspondientes *snippets* en la tarea en ejecución.

- **Librería de monitorización dinámica (DMLib).**

  La DMLib tiene como objetivo facilitar la instrumentación y recolección de datos de rendimiento. Es una librería compartida cargada de manera dinámica por el Controlador de Aplicación en las tareas que componen la aplicación paralela. Para realizar su objetivo, la librería contiene funciones responsables del registro de los eventos con todos los atributos requeridos, así como funciones encargadas del envío de dichos eventos para el análisis.

- **Analizador.**

  Es un proceso que realiza el análisis de rendimiento de la aplicación paralela, detectando automáticamente los problemas de rendimiento existentes y solicitando los cambios necesarios para mejorar el rendimiento de la aplicación.

En las siguientes secciones se describen con detalle todos los módulos que componen MATE, presentando su funcionalidad, requerimientos y limitaciones.

3.3.1 Controlador de aplicación

Como se introdujo anteriormente, cada controlador de aplicación es un único proceso que controla una única tarea MPI ejecutándose en una máquina local. Este proceso proporciona los siguientes servicios:

- Control distribuido de la aplicación.
    - Inicia y finaliza cada tarea MPI.

- Gestión de la instrumentación de la aplicación.
    - Gestiona la instrumentación de las tareas en ejecución.
    - Permite remotamente al Analizador añadir/eliminar instrumentación.

- Monitorización del rendimiento.
    - Carga la librería de monitorización compartida en las tareas de la aplicación.
    - Genera los snippets de monitorización.
    - Inserta/elimina los snippets.

- Sintonización del rendimiento.
    - Carga la librería de sintonización compartida en las tareas de la aplicación.
    - Genera los snippets de sintonización.
    - Inserta/elimina los snippets.

El Controlador de Aplicación está compuesto por varios módulos que cooperan entre sí, los cuales se muestran en la figura 3.3.

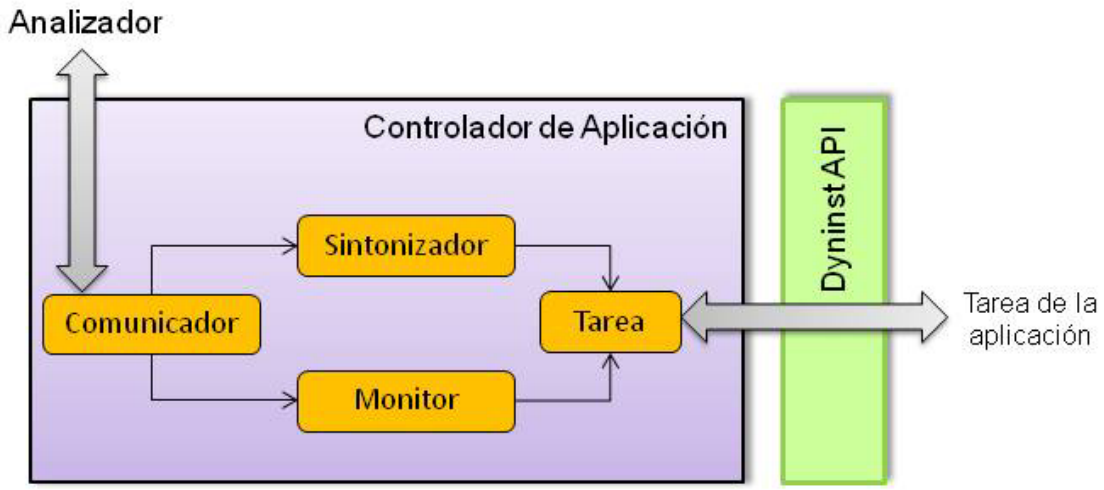

Figura 3.3. Arquitectura interna del Controlador de Aplicación

El comunicador es el módulo del Controlador de Aplicación que gestiona la comunicación con el mundo exterior usando el protocolo TCP/IP. Éste despacha los mensajes que le llegan hacia componente correspondiente, Monitor o Sintonizador, los cuales actuarán de la manera indicada sobre la tarea de la aplicación mediante la API proporcionada por Dyninst. Más detalles pueden ser encontrados en [7].

Los módulos Monitor y Sintonizador son los que mayor funcionalidad presenta dentro del Controlador de Aplicación ya que participan en las fases claves del proceso de sintonización. En las siguientes secciones se describen con más detalle estos componentes.

Monitor

El Monitor es el módulo responsable de la monitorización de la ejecución de la aplicación. La monitorización está basada en eventos que se generan mediante llamadas a funciones. La aplicación es instrumentada dinámicamente en tiempo de ejecución y la instrumentación insertada genera eventos. Cuando MATE es lanzado, el Analizador indica al Monitor el conjunto de eventos que deben ser trazados. Conceptualmente, estos eventos reciben el nombre de puntos de medida. De este modo, cuando la aplicación comienza su ejecución, el Monitor inserta el código necesario para capturar los eventos en la aplicación en ejecución.

El Monitor ofrece una API que permite el Analizador añadir o eliminar dinámicamente un evento. En la API un evento queda definido mediante el identificador del proceso donde se añade/elimina, el identificador de dicho evento, el nombre de la función en la que se generará, el punto del código que determina donde el evento sería generado, el número de atributos que se almacenarían en dicho evento y las propiedades de tales atributos.

Para realizar la traza dinámica de eventos, el Monitor usa la librería Dyninst para insertar el código de instrumentación, *snippet,* que genera eventos para ser trazados. Para recopilar estos eventos y enviarlos al Analizador, el Monitor usa la DMLib cargada en cada tarea durante el proceso de arranque.

La instrumentación puede variar durante la ejecución. Para encontrar cuellos de botella el Analizador puede necesitar alguna información adicional, o puede necesitar eliminar alguna instrumentación que no utilice habitualmente. Cuando ocurre esto, el Analizador notifica al Monitor y como consecuencia éste último modifica el conjunto de eventos monitorizados. Es de destacar que la comunicación entre el Monitor y el Analizador es establecida usando un protocolo de bajo nivel de recolección de eventos basado en TCP/IP.

Sintonizador

El Sintonizador es el módulo responsable de aplicar las acciones de sintonización sobre las tareas de la aplicación. Los cambios necesarios son determinados por las soluciones propuestas por el Analizador a partir de las cuales el Sintonizador modifica la aplicación empleando de nuevo Dyninst, cambiando la memoria asociada a la misma.

Tras la fase de análisis y para realizar la sintonización de la aplicación, el Analizador puede requerir de un conjunto de acciones que le permite llevar a cabo la modificación de los

parámetros críticos de forma correcta. Estas acciones puede ser *cargar una librería*, *establecer el valor de una variable*, *insertar la llamada a una función*, etc. Para llevar a cabo este conjunto de acciones el Sintonizador proporciona una API al Analizador a través de la cual llevarlas a cabo.

Cada una de las acciones de sintonización incluye un parámetro de sincronización o *breakpoint*, el cual se inserta en un punto determinado de la aplicación, de modo que determina cuando la acción de sintonización podría ejecutarse para asegurar que el comportamiento de la aplicación siga siendo correcto. Cuando la ejecución de la aplicación alcanza el *breakpoint*, la acción de sintonización se ejecuta y el *breakpoint* queda eliminado.

3.3.2 Librería de Monitorización Dinámica (DMLib)

DMLib es una librería dinámica que proporciona la funcionalidad necesaria para realizar la traza de eventos y está implementada como una librería compartida. El Controlador de Aplicación carga esta librería en el espacio de direcciones de cada proceso de la aplicación para simplificar la instrumentación y recolección de datos.

Esta librería ofrece una API que contiene funciones que son responsables del registro de eventos con todos los atributos requeridos y de su envío para el análisis:

- *Inicializar la librería*, proporcionando información sobre el proceso que va a ser monitorizado y la localización del host donde se aloja el Analizador.
- *Finalizar la librería*, con el objetivo de liberar todos los recursos adquiridos y notificar al Analizador que los procesos de aplicación han terminado y cerrado la conexión con él.
- *Registrar eventos*. En este caso el identificador o nombre del evento y los atributos del mismo deben ser proporcionados, así como la función o punto específico en el cual el evento seria capturado. Cuando el registro de un evento finaliza, significa que está preparado para ser enviado al Analizador.

Para evitar la sobrecarga de la red, la implementación de DMLib usa un mecanismo de buffering para gestionar los eventos. En lugar de enviar cada evento de manera individual, existe un buffer interno usado para agrupar eventos y enviarlos en mensajes de tamaño más elevado. Esto permite la reducción del número de mensajes generados y limita la intrusión. Dicho envío es controlado mediante marcas de tiempo con el fin de evitar una espera excesiva en la cola de envío para eventos individuales.

3.3.3 Analizador

El Analizador es el módulo que dirige la sintonización de la aplicación. Para ello solicita las métricas necesarias, que le permiten llevar a cabo el análisis de rendimiento, e indica los cambios en la aplicación.

Para ser capaz de evaluar al comportamiento de una aplicación dada, el Analizador necesita algún tipo de conocimiento sobre la aplicación y traza de eventos online. Desde un punto de vista funcional, el Analizador se divide en dos módulos diferenciados: la API de Sintonización Dinámica (DTAPI) y los tunlets.

API de Sintonización Dinámica (DTAPI)

Esta API presenta la funcionalidad que permite gestionar el proceso de mejora de rendimiento de la aplicación. En ella se encuentran todos los aspectos de bajo nivel relacionados con la administración de los eventos entrantes, la gestión necesaria para el comienzo y la terminación de una tarea, información descriptiva de las tareas en ejecución de la aplicación, la información necesaria para sintonizarlas, etc. Como se explicará en la siguiente sección los tunlets usan DTAPI como interfaz a través de la cual establecen la instrumentación necesaria para evaluar el modelo de rendimiento.

Esta librería es implementada como un sistema distribuido y asíncrono donde:

- Las peticiones de monitorización y sintonización son delegadas en los Controladores de Aplicación distribuidos que en su lugar instrumenta y sintoniza las tareas de la aplicación.
- Los eventos recibidos procedentes de la librería de Sintonización Dinámica y los Controladores de Aplicación son recopilados y despachados hacia los manejadores de eventos.

Tunlets

Los tunlets son el núcleo de la sintonización dinámica y automática implementada por MATE, en términos de representación del conocimiento. Cada tunlet define e implementa una particular técnica de sintonización, por ejemplo, la lógica necesaria para superar un particular problema de rendimiento mediante la encapsulación del conocimiento sobre el problema de rendimiento en los siguientes términos:

- Un conjunto de **puntos de medida**, los cuales indican que es necesario medir en la aplicación para evaluar su comportamiento. Esta definición incluye valores de variables, parámetros, marcas de tiempo, etc.

- Un conjunto de **funciones de rendimiento**, que son expresiones matemáticas que determinan como evaluar la información recopilada para detectar cuellos de botella.
- Un conjunto de **acciones de sintonización,** que indican *que*, *donde* y *cuando* cambiar en la ejecución de la aplicación con el objetivo de adaptar su comportamiento.

Los tunlets usan la API de Sintonización Dinámica para dirigir el proceso de análisis de rendimiento de la aplicación. Al inicio del proceso de sintonización el tunlet mediante la API, indica el conjunto de eventos de monitorización que deben ser insertados en una determinada tarea. Cuando el mensaje de la generación de un determinado evento llega al Analizador, es redirigido al tunlet, el cual analiza los parámetros existentes en dicho evento que describen el comportamiento de la aplicación. Cuando el tunlet detecta un posible problema de rendimiento usa la DTAPI para cambiar algún tipo de instrumentación realizada anteriormente o bien realizar la modificación de algún parámetro crítico para mejorar el rendimiento. Cuando el proceso de sintonización termina, el tunlet finaliza y se descarga de la memoria.

La DTAPI se dispone como un conjunto de clases C++, que los tunlets emplean como interfaz para trabajar correctamente integrados en MATE.

3.4 Metodología de funcionamiento

La figura 3.4 muestra la ejecución de una aplicación bajo MATE, identificando los distintos componentes de su arquitectura que participan y la funcionalidad de los mismos.

Cuando la ejecución de la aplicación comienza, un particular tunlet indica al Analizador cual es el conjunto de puntos de medida necesarios. El Analizador envía estos requerimientos a cada Controlador de Aplicación, concretamente al módulo Monitor, los cuales se encuentran distribuidos con cada tarea de la aplicación paralela. Cuando la ejecución de la aplicación o de alguna tarea en concreto alcanza un punto de instrumentación, se produce la creación de un evento. El evento posee determinados atributos que lo caracterizan, los cuales contienen información relacionada con el entorno de ejecución y con la aplicación, como puede ser marcas de tiempo, valores de variables, etc. Esta información es clave para poder calcular los parámetros de rendimiento del modelo matemático y evaluar las expresiones de rendimiento asociadas. Una vez generado el evento, la DMLib lo envía junto con sus atributos al Analizador.

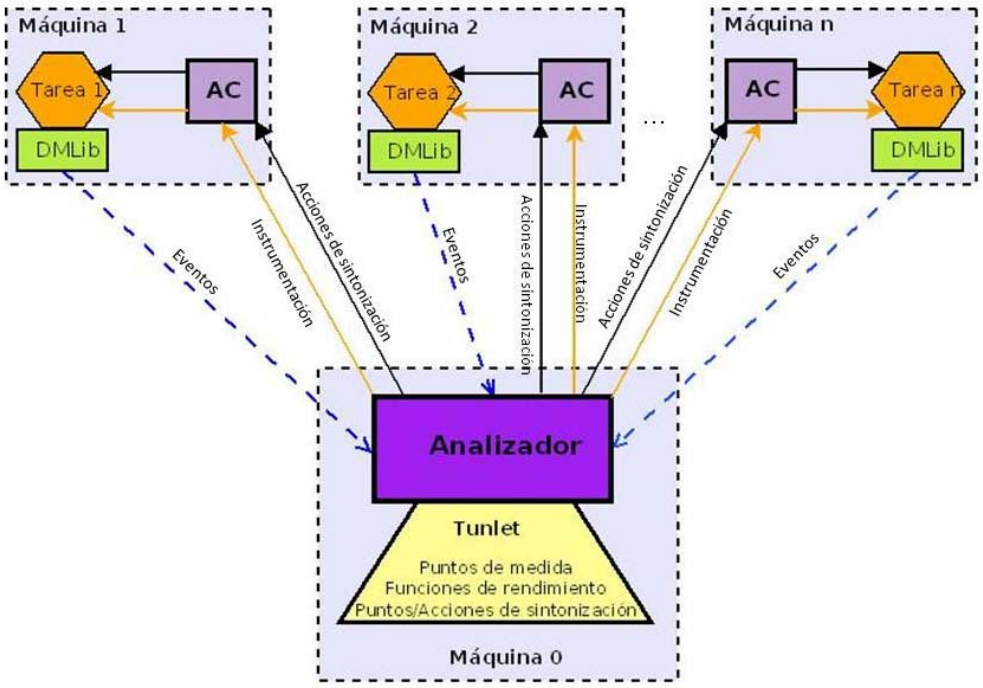

Figura 3.4 Arquitectura de MATE sintonizando dinámicamente para MPI

A lo largo de la ejecución, el Analizador va recibiendo los eventos solicitados procedentes de los distintos procesos de la aplicación paralela, lo cual se le notifica al tunlet correspondiente. Los eventos recibidos son clasificados de acuerdo a un tipo en concreto definido en el tunlet.

Cuando el tunlet recibe mediante los eventos toda la información necesaria, evalúa las funciones de rendimiento y determina el rendimiento actual y óptimo. Si el tunlet detecta un cuello de botella, decide si el actual rendimiento puede ser mejorado bajo las actuales condiciones. En caso positivo, el tunlet informa al Analizador de los posibles cambios necesarios, y en consecuencia éste solicita las correspondientes acciones de sintonización. La solicitud determina que debería ser cambiado (punto/acción/sincronización de sintonización) y esto es enviado a la instancia del Controlador de Aplicación correspondiente, en concreto es reenviado al módulo Sintonizador.

Como se muestra conceptualmente en la figura 3.4, cada tunlet proporciona los elementos para dirigir las fases de monitorización, análisis y sintonización. Es de destacar, que los cambios que se realizan en la aplicación paralela en tiempo de ejecución, en los procesos de monitorización y sintonización, son implementados mediante la librería de instrumentación dinámica Dyninst.

3.5 MATE y otras herramientas de sintonización dinámica

MATE presenta una aproximación para la sintonización dinámica del rendimiento en aplicaciones paralelas. Esta aproximación se fundamenta en los principios empleados en técnicas de análisis dinámico de rendimiento basadas en la instrumentación dinámica mediante el uso de la librería Dyninst [8], como la desarrollada en la herramienta Paradyn [33].

MATE comparte algunas características con las herramientas de sintonización anteriormente comentadas en el capítulo 2, aunque también existen aspectos que las diferencian. En lo referente a los entornos de ejecución, MATE inicialmente estaba pensada para sintonizar aplicaciones PVM paralelas/distribuidas desarrolladas en C/C++ ejecutándose en plataformas UNIX y actualmente está siendo desarrollada para sintonizar aplicaciones basadas en MPI. También se desarrolló en 2008 una versión orientada para entornos Grid, denominada GMATE [43]. Por su parte, Autopilot [35] y PerCo [37] han sido diseñadas con el fin de ser empleadas especialmente en sistemas de computación heterogéneos.

En el proceso de monitorización, MATE monitoriza la aplicación mediante la instrumentación dinámica insertando puntos de medida los cuales generan eventos que serán enviados al analizador. De forma semejante a MATE en Autopilot, el proceso de monitorización se basa en la inserción dinámica de sensores, los cuales son procesos que permiten extraer información de rendimiento de la aplicación sintonizada. En el caso de Active Harmony [36], se realiza una continua monitorización del rendimiento de las distintas implementaciones de librerías subyacentes, de modo que basándose en la información recopilada, se redireccionan las llamadas de las funciones del programa a las de la librería subyacente seleccionada.

Por otro lado, en lo relacionado con el análisis de rendimiento, MATE usa reglas normales y modelos de rendimiento en los cuales se formalizan posibles problemas de rendimiento. Mientras que Autopilot emplea lógica difusa para automatizar el proceso de toma de decisiones, Active Harmony usa técnicas heurísticas mediante las cuales explora el espacio de optimización del patrón de la aplicación teniendo en cuenta las librerías subyacentes y ajusta los valores en la sintonización basándose en el rendimiento observado. PerCo basa su análisis en el empleo de técnicas para predecir el rendimiento, combinando series de tiempo y métodos de ajuste de datos. Las series de tiempo generan predicciones aplicando una fórmula matemática al histórico de datos; por su parte el ajuste de datos es aplicado a los datos históricos para obtener una fórmula matemática con muchas variables, de manera que evaluando dicha fórmula con los valores actuales (número de procesadores, tamaño de los datos de la aplicación, etc) se generan las predicciones.

Finalmente, si se considera la preparación de la aplicación para la sintonización, usando MATE la sintonización se basa en la instrumentación dinámica donde los puntos de sintonización son determinados por el modelo de rendimiento; éstos últimos actualizando su valor sobre la marcha cuando el analizador lo determina. Para realizar esta acción en ocasiones es necesario adaptar la aplicación o poseer un mayor conocimiento de la aplicación que va a ser sintonizada. Igual que en MATE, en Autopilot los actuadores son insertados de manera dinámica en el proceso de sintonización pero para ello se requiere conocimiento sobre la aplicación. En Active Harmony el mecanismo se basa en la integración y elección de diferentes librerías con la misma funcionalidad. Y en PerCo, en el proceso de sintonización no se requiere la interacción directa del usuario ya que las decisiones de reimplementación se basan en una política que ha sido construida en el módulo que dirige el rendimiento de la aplicación.

3.6 Limitaciones de MATE como entorno de sintonización

En MATE, el análisis de rendimiento es realizado de forma centralizada, ya que existe un único Analizador responsable de recibir y procesar todos los eventos que le llegan directamente desde la aplicación, más específicamente desde las Librerías de Monitorización Dinámica asociadas a cada tarea MPI, tal y como se explico en la sección 3.4. El Analizador es ejecutado en una máquina independiente para reducir la sobrecarga causada por el continuo proceso de análisis en máquinas donde la aplicación se está ejecutando.

Sin embargo, aunque esta aproximación centralizada funciona, presenta algunos problemas relacionados con dos factores diferentes:

- El número de máquinas involucradas en la ejecución de la aplicación.
- La persistencia de problemas de rendimiento.

Con respecto al primero de los factores, se puede asumir que conforme el número de tareas involucradas en la aplicación aumenta, el número de eventos también se incrementa de manera proporcional. Como consecuencia el Analizador se convierte en un cuello de botella que afecta a la efectividad del sistema.

El Analizador posee una hebra que recolecta los eventos recibidos desde la aplicación. Independientemente de la cantidad de eventos entrantes, éstos son gestionados siguiendo una política FIFO (*first in, first out*). En consecuencia, el tiempo gastado en procesar la información es proporcional a la cantidad de eventos. Además, en ocasiones ocurren *olas* de eventos; esto quiere decir que cada tarea de la aplicación genera eventos aproximadamente al mismo tiempo, causando la sobrecarga del analizador en unos instantes determinados, tales como el final de la

iteración, cuando todos los procesos de la aplicación terminan. Además, mientras el Analizador está procesando datos de la iteración *i* continúa su ejecución por la iteración *i+1*.

En lo relacionado con la persistencia de problemas de rendimiento, se debería tener en cuenta que la sintonización dinámica está basada en asumir que los problemas de rendimiento surgen en más de una iteración. Esta es la razón de porqué el análisis de rendimiento para detectar problemas y encontrar soluciones debería ser rápido. En el caso de MATE, el análisis queda reducido a la evaluación de un conjunto de expresiones analíticas. Sin embargo, para evaluar estas expresiones, es necesario procesar todos los eventos que llegan para obtener los valores de los parámetros del modelo de rendimiento. De forma similar, cuando el número de eventos crece, el tiempo de procesamiento de la información asociada aumenta también.

Por lo tanto, si consideramos estas dos situaciones al mismo tiempo, el cuello de botella causado por la recolección de eventos y su posterior procesamiento puede significar que cuando la solución para un problema existente esté lista para ser insertada, tal vez el problema de rendimiento haya cambiado o desaparecido.

Se concluye que estos problemas limitan las propiedades de escalabilidad de MATE. Por ello, con el objetivo de aumentar la usabilidad de MATE, en el trabajo expuesto en [45] [46] se realiza un estudio para proporcionar a MATE la escalabilidad necesaria. Se propone una nueva aproximación de recolección y preprocesado de eventos jerárquica-distribuida, cuyo objetivo es resolver el cuello de botella que significa el Analizador. Esta aproximación se basa en la distribución de la recolección de eventos lo cual disminuye la sobrecarga con respecto a la manera original centralizada en la que dicha recolección era llevada a cabo, y el preprocesado de operaciones acumulativas o de comparación siempre que sea posible.

Los resultados de este trabajo muestran la resolución del cuello de botella que presentaba la aproximación centralizada empleada por MATE originalmente. Además se presenta una mejora en la sincronización del proceso de análisis con la ejecución de la aplicación ya que la estructura jerárquica de recolección de eventos aumenta la probabilidad de detectar y procesar cada evento tan rápido como es recibido. Este hecho logra además un decremento en el tiempo de procesamiento del Analizador debido al preprocesado de eventos realizado. De tal modo, las ideas establecidas en este trabajo, suponen una buena base de partida para escalar el proceso de análisis en MATE.

Capítulo 4

Modelo de rendimiento para aplicaciones Master/Worker

4.1 Introducción

La predicción de rendimiento es un aspecto importante para conseguir ejecuciones eficientes en programas paralelos. Conseguir un buen rendimiento de un código paralelo es una tarea ardua y difícil debido a la complejidad de los sistemas multiprocesador y a las dificultades en el análisis de su rendimiento. Por ello, la predicción del rendimiento es una utilidad esencial para la depuración de programas paralelos ya que ofrece información interesante para aumentar la eficiencia de los mismos.

Actualmente existen varias alternativas para obtener un modelo de rendimiento de las aplicaciones usando diferentes aproximaciones. En general estas aproximaciones se pueden clasificar en tres categorías:

- **Modelado por simulación.**

  Un simulador, construido como una aplicación software, es un sistema completo que emula el comportamiento de cada uno de los subsistemas de una arquitectura paralela. No sólo emula el comportamiento temporal de cómo se ejecuta en él un determinado algoritmo, sino que también es capaz de extraer otros parámetros de la

arquitectura del sistema multicomputador: fallos de caché, instrucciones de procesador ejecutadas, parámetros de la red de interconexión, etc.

Esta técnica de evaluación de rendimiento está especialmente indicada para desarrollar nuevas arquitecturas paralelas ya que permite observar el comportamiento de aplicaciones en sistemas que todavía no han sido implementados. Sin embargo, simular el comportamiento de aplicaciones enteras puede ser muy costoso y por tanto no es adecuado para utilizar estos modelos en herramientas de evaluación de rendimiento interactivas.

- **Modelado analítico.**

  La idea básica de los modelos analíticos consiste en modelar tanto la arquitectura paralela como el algoritmo usando métodos analíticos. El programa paralelo que está siendo desarrollado puede analizarse independientemente de la arquitectura en que va a ser implementado, lo que permite incluso analizar las posibles arquitecturas o sistemas futuros que todavía estén en diseño y que podamos modelar por una serie de parámetros.

  Los modelos analíticos son métodos rápidos y efectivos comparados con otras técnicas de modelado, ya que utilizan soluciones eficientes basadas en ecuaciones matemáticas. Sin embargo, el grado de fiabilidad con respecto a la realidad puede quedar mermado por las características de los parámetros elegidos para el modelo y es inherente a las suposiciones y simplificaciones que se hacen del sistema paralelo y del algoritmo.

- **Modelado por obtención de métricas y trazas.**

  La mejor forma de obtener resultados precisos para modelar una aplicación es medir su comportamiento en un sistema paralelo real. Analizando los datos de traza obtenidos, el usuario puede identificar y corregir los cuellos de botella en la aplicación, pero esto significa tener a disposición un sistema paralelo para realizar el desarrollo de los programas. Además se necesitan herramientas específicas de medida como librerías de instrumentación y herramientas de análisis de traza, como las expuestas en la sección 2.21 del capítulo 2. Estas técnicas se usan normalmente para extraer los parámetros que luego se utilizarán en los modelos analíticos, en las simulaciones, para validar un modelo de rendimiento determinado, etc.

En el presente trabajo de investigación se ha estudiado el modelo de rendimiento expuesto en [9] para aplicaciones desarrolladas bajo un paradigma Master/Worker. Se trata de un modelo analítico cuyos parámetros corresponden a medidas de rendimiento que determinan

el comportamiento de la aplicación en un sistema paralelo real; tiene como objetivo mejorar el rendimiento de aplicaciones construidas bajo el citado paradigma atacando los problemas de rendimiento que las caracterizan.

Las aplicaciones Master/Worker pueden sufrir dos cuellos de botella de rendimiento relacionados con su estructura y funcionalidad: el primero de ellos es el desbalanceo de carga de los workers, lo cual puede producir largos periodos de inactividad para workers rápidos o bajamente cargados; y el segundo es el uso de un inadecuado número de workers para procesar el conjunto de tareas. De este modo, el modelo de rendimiento estudiado intenta solventar los citados problemas siguiendo una estrategia de actuación sobre la aplicación basada en 2 fases: una primera fase que emplea una estrategia para balancear la carga de manera que los recursos se usan de forma eficiente y una segunda para predecir el número de workers más adecuado para mejorar el rendimiento de la aplicación.

Una vez introducida la problemática y conceptos principales, en las siguientes secciones se expone de manera detallada el modelo de rendimiento para aplicaciones Master/Worker. En concreto se describe aquellas expresiones y conocimientos que se han aplicado directamente en el presente trabajo de investigación. Primeramente se planteará la estrategia seguida para alcanzar el balanceo de la carga computacional y a continuación se describirá los índices empleados para predecir el número de workers que optimizan tiempo de ejecución y uso de recursos.

4.2 Balanceo de la carga entre los workers

El tiempo de ejecución para una aplicación Master/Worker con N workers y un conjunto de tareas que pueden ser secuencialmente procesadas en un tiempo T, se encuentra limitado en un rango comprendido entre T/N (límite inferior), y T (límite superior), teniendo en cuenta que en dichas expresiones queda omitido el tiempo de comunicación. De tal modo, que conseguir un tiempo de ejecución cercano al límite inferior depende principalmente de la existencia de un buen balanceo de carga entre los workers, lo cual a su vez se basa en una buena política de distribución de datos.

En general las técnicas de balanceo de carga intentan compensar el desbalanceo asignando más trabajo a aquellos workers que terminan su trabajo antes. Para lograr el balanceo estas técnicas realizan una distribución parcial dividiendo el conjunto de tareas en diferentes porciones llamadas batches, cuyo tamaño queda determinado por un factor de partición. De tal modo, que el número de tareas asignadas a cada batch depende de la estrategia de distribución seguida, y es posible que sea diferente de un batch a otro. La idea es distribuir el primero de esos batches entre los workers en trozos o chunks del mismo tamaño, y cuando un worker

termine el procesamiento de su chunk asignado el master le enviará un nuevo chunk del Batch que corresponda. Este proceso, mostrado en la figura 4.1, continúa hasta que todos los batches han sido distribuidos. Partiendo de esta idea se obtiene que aquellos workers que han recibido tareas más pesadas no reciban más carga y workers que han recibido tareas más ligeras sean empleados para realizar más trabajo.

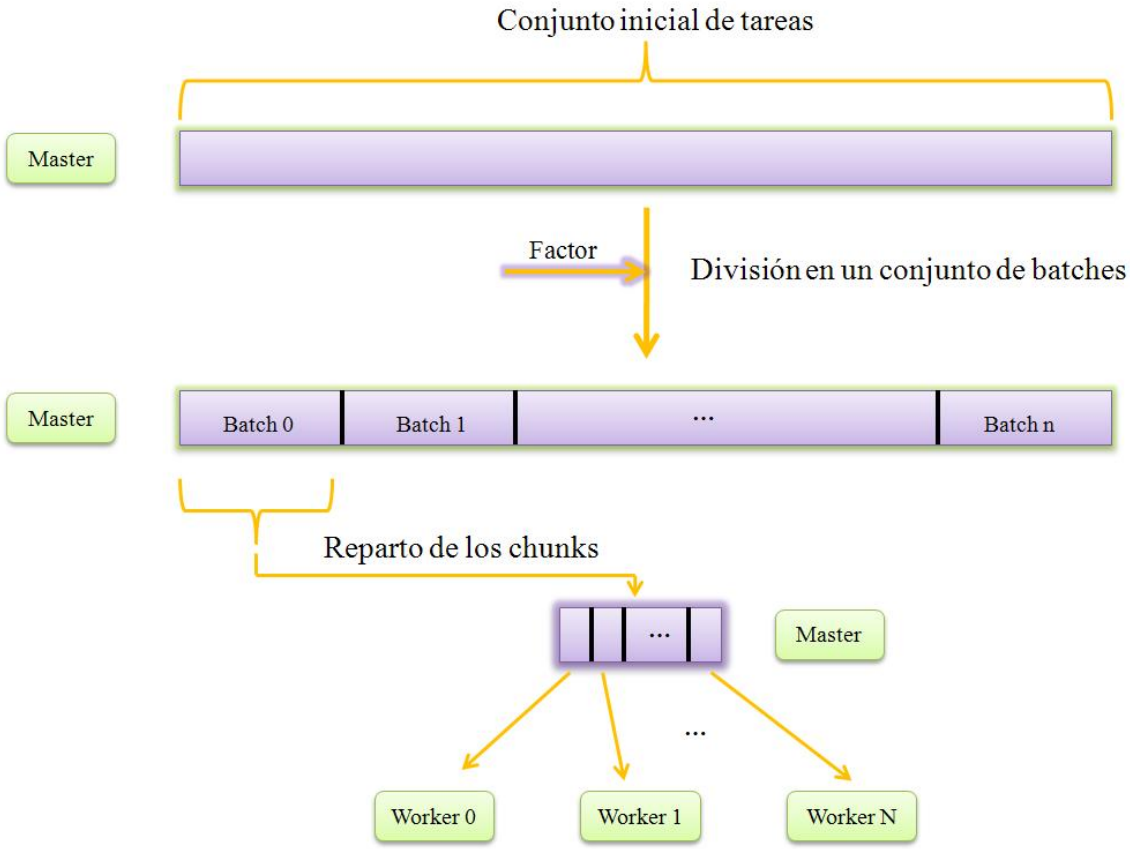

Figura 4.1 Técnica de balanceo de carga

Existen diferentes estrategias para determinar el tamaño de un batch con el propósito de obtener un buen balanceo de carga con costes mínimos en computación y comunicación. En el trabajo estudiado expuesto en [9] se plantean 3 principales técnicas de balanceo de carga:

- *Fixed Size Chunking* (*FSC*), el cual consiste en dividir el conjunto de tareas en un determinado número de batches de igual tamaño. En este caso, para una aplicación en concreto, se debe intentar encontrar cuál es el mejor número de batches para mejorar el balanceo de carga.

- *Dynamic Predictive Factoring* (*DPF*), parte de la idea de construir el primer batch con alguna porción del conjunto de tareas, el segundo batch con la misma porción de las tareas restantes, y así de manera sucesiva hasta alcanzar un límite fijado en el tamaño de los batches.

- *Dynamic Adjusting Factoring* (*DAF*), presenta las mismas características que *DPF* pero empleando un factor variable para ir calculando el tamaño de los batches teniendo en cuenta las condiciones actuales de carga en la aplicación.

En el presente trabajo de investigación se ha implementado *DAF* como estrategia para alcanzar el balanceo de carga dentro del modelo de mejora de rendimiento para aplicaciones Master/Worker.

4.2.1 Dynamic Adjusting Factoring (DAF)

Dynamic Adjusting Factoring [47] es una política de distribución de datos que dinámicamente intenta adaptar el factor de partición a las actuales condiciones de la aplicación. La política de *Factoring* original [48] en la que se basa, destinada a la gestión de bucles paralelos, intenta asignar a los procesadores los chunks más grandes posibles de las iteraciones paralelas del bucle minimizando la probabilidad de exceder el tiempo de ejecución óptimo (*T/N*).

Este *Factoring* para bucles paralelos asume que el tiempo de ejecución de un chunk es una variable aleatoria obtenida como la suma de variables aleatoria idénticas, cada una de las cuales representa una iteración del bucle paralelo. Este modelo original tiene en cuenta el comportamiento independiente de todas las iteraciones porque su tiempo de ejecución puede ser determinado.

Este modelo puede ser fácilmente adaptado a aplicaciones Master/Worker sustituyendo las iteraciones del bucle paralelo por tareas. En este caso, la variable aleatoria queda definida como el tiempo de procesamiento de una tarea (*C*), el cual es inferido dividiendo el tiempo de ejecución de un chunk entre el número de tareas presentes en ese chunk.

La formulación matemática de este modelo asume que el entorno presenta *N* workers disponibles para ejecutar *M* tareas (*M*>>*N*), cada una de las cuales queda modelada mediante una variable aleatoria *C* caracterizada por sus parámetros estadísticos de media (μ_C) y desviación estándar (σ_C). Además se considera que el entorno es homogéneo, todos los procesadores tendrán la misma media y desviación estándar del tiempo de procesamiento de tarea.

Aceptando que inicialmente todos los workers están ociosos, el tiempo de ejecución de *N* chunks de F_0 tareas en paralelo puede ser modelado por un estadístico de orden *N*, siendo F_0 el número de tareas asignadas en cada chunk del primer batch. El valor esperado para el estadístico de orden *P* para cualquier distribución de media μ y desviación estándar σ se encuentra limitada por la expresión:

$$\mu + \sigma\left((N-1)/\sqrt{2N-1}\right) \leq \mu + \sigma\sqrt{N/2} \qquad (4.1)$$

Si el primer batch tiene chunk de tamaño F_0 tareas, el tiempo de ejecución de un worker puede ser modelado como una nueva variable aleatoria obtenida por el producto de la variable aleatoria del tiempo de procesamiento por tarea, y el número de tareas F0. Esta nueva variable tendrá una media $\mu_C F_0$ y una desviación estándar $\sigma_C F_0$. Así, el valor esperado del estadístico de N chunks paralelos (cada uno con F_0 tareas) en el primer batch es:

$$\mu_C F_0 + \sigma_C F_0 \sqrt{N/2} \tag{4.2}$$

Siendo el propósito no superar el tiempo óptimo de ejecución, el cual viene expresado por $\mu_C(M/N)$, que se cumpla la condición impuesta por la expresión $\mu_C F_0 + \sigma_C F_0 \sqrt{N/2} = \mu_C(N/2)$ es el objetivo de la política de ajuste dinámico para el número de workers. Para alcanzar ese fin, se necesita calcular F_0, el cual sería la porción del conjunto completo de tareas a ser distribuidas en el primer batch dividido entre el número de procesos, $M/(x_0 N)$, donde x_0 es el inverso del factor de partición usado por la política para generar el primer batch a ser distribuido. Se obtiene que:

$$x_0 = 1 + \left(\sigma_C \sqrt{N/2}\right)/\mu_C \tag{4.3}$$

Para el cálculo de los siguientes batches, los chunks presentan un tiempo de ejecución de media $\mu_C F_j$ y desviación estándar $\sigma_C F_j$, pero ahora se necesita otra aproximación ya que no se puede asumir que todos los workers están ociosos y por tanto el tiempo de comienzo de cada chunk es distinto. Tal y como se expone en la política de Factoring original [48], el número de tasks F_j se determina partiendo de la idea de que queda suficiente trabajo para solventar el desbalanceo. De aquí se deduce las siguientes expresiones que permiten obtener x_j, es decir, el inverso del factor de partición para calcular el resto de los batches:

$$\mu_C F_j + \sigma_C F_j \sqrt{N/2} = \mu_C \left(\left(\frac{R_j}{N} \right) - F_j \right)$$

$$F_j = \frac{R_j}{x_j N}$$

$$x_j = 2 + \left(\sigma_C \sqrt{N/2}\right)/\mu_C \tag{4.4}$$

Finalmente, se obtiene que mediante el uso de las ecuaciones (4.1) y (4.2) y empleando una estrategia de sintonización dinámica es posible adaptar a las actuales condiciones de la aplicación el factor de partición que determina el tamaño de los batches a lo largo de la iteración.

El algoritmo de distribución que se propone en [9] y que será el implementado en el presente trabajo de investigación corresponde a los siguientes pasos:

1. Con el objetivo de acumular suficiente información para calcular el factor adaptativo de distribución, la primera iteración de la aplicación es ejecutada empleando un factor fijo de 0.5 para la creación de todos sus batches. Este valor inicial de factor es empíricamente escogido porque en general es el que mejor comportamiento presenta.
2. Al principio de las restantes iteraciones se calcula x_0, mediante la ecuación 4.1 y empleando la información empleada en el pasado. De tal modo, a partir de x_0 se obtiene el valor del factor adaptativo para el primer batch. Seguidamente, empleando los mismos datos históricos y la ecuación 4.2, calculamos x_1 y por tanto el factor adaptativo para el batch *1*. De tal modo que los batchs *0* y *1* ya están preparados para ser distribuidos.
3. A lo largo del proceso de distribución de tareas, cuando el número de chunks disponibles del batch que actualmente está siendo repartido cae por debajo de un umbral definido (el cual en nuestro trabajo ha sido fijado a la mitad del número de workers), se usa la ecuación 4.2 para calcular x_j, de manera que se obtiene el número de tareas del batch *j* y éste se encuentra preparado para su distribución.
4. Si el número de tareas por chunk alcanza un límite mínimo predefinido, las restantes tareas son distribuidas entre los últimos chunks creados, y el proceso de distribución finalizada.

Tal y como se deduce de la lógica del algoritmo presentado, los valores correspondientes a los tiempos de cómputo por tarea empleados para calcular x_j se deben de ir acumulando a lo largo de la iteración y de una iteración a otra con el objetivo de que los valores de los factores de distribución se vayan adaptando a las condiciones de balanceo que presenta la aplicación.

En el presente trabajo de investigación, con el objetivo de que en los factores de distribución no influyan iteraciones previas en las cuales el estado de la aplicación era distinto al que se presenta en la iteración actual, se ha definido una ventana que delimita el histórico de acumulación de los tiempos de cómputo comentados.

4.3 Determinación del número de workers

Como se comentó en la sección 4.2, en una aplicación Master/Worker ideal el tiempo total de ejecución sería igual el tiempo secuencial de ejecución dividido por el número de workers. Esto hecho ocurriría si:

- No hubiese coste de comunicación.
- La aplicación se está ejecutando sobre una plataforma dedicada y homogénea.
- Se ha alcanzado un balanceo de la carga.
- La computación escala computacionalmente.

En el entorno de computación definido por las anteriores características, cualquier recurso disponible que pueda ser asignado a la aplicación debe serlo, porque será eficientemente usado para mejorar el rendimiento. Sin embargo en el mundo real, se observa que el speedup de una aplicación usualmente decrece cuando se añaden nuevos recursos, poniendo de manifiesto una pérdida de eficiencia, ya que los costes introducidos (aumento del volumen de comunicaciones) son mayores que las ventajas que los nuevos recursos proporcionan.

Consecuentemente el modelo de rendimiento estudiado, tiene en cuenta todos estos parámetros e intenta evaluar el comportamiento de la aplicación cuando se está ejecutando y decidir si merece la pena cambiar el número de workers para mejorar el rendimiento de la misma. En el desarrollo de dicho modelo se ha asumido que existe un solo worker por elemento de procesamiento, y que la aplicación está balanceada. El desarrollo detallado del modelo así como su discusión, se muestra y comentan en [9].

En nuestro trabajo de investigación para determinar y sintonizar el número de workers de la aplicación, se ha empleado un índice de rendimiento *Pi* que directamente permite relacionar rendimiento con eficiencia en el uso de recursos. La principal ventaja de este índice es que puede ser automáticamente optimizado porque permite encontrar la mejor relación posible entre eficiencia y ganancia de rendimiento.

La eficiencia de una aplicación se encuentra definida como la porción de tiempo que los workers están realizando trabajo útil sobre el tiempo que éstos han estado disponibles para realizar trabajo provechoso. Más formalmente, para *x* workers el índice de eficiencia *E(x)* queda definido como $\frac{T_c}{T_{avail}}$, donde T_c es el tiempo de procesamiento total de todos los workers, T_{avail} es $\sum_{i=0}^{x-1} tavail_i$, y $tavail_i$ es el tiempo que el worker *i* ha estado disponible para hacer trabajo útil, lo cual para una aplicación como las que se pretende modelar, será el tiempo de una iteración completa (T_t). De tal modo, el índice de eficiencia para *x* workers viene dado por la siguiente expresión:

$$E(x) = T_c/xT_t(x). \tag{4.5}$$

Consecuentemente, el índice de rendimiento para x workers sería:

$$Pi(x) = T_t(x)/E(x) = xT_t(x)^2/T_c \tag{4.6}$$

Basándonos en las expresiones definidas en el modelo de rendimiento para describir el comportamiento de una aplicación Master/Worker, y suponiendo que el protocolo de comunicación de la aplicación a modelar es asíncrono (el master realiza una operación de envío y los datos son almacenados en un buffer intermedio de forma que el siguiente envío puede realizarse antes que el envío previo haya finalizado), el tiempo de una iteración completa viene definido por la siguiente expresión:

$$T_t(x) = 2m_o + \left[\left((x-1)\alpha + 1\right)\lambda V + T_c\right]/x \tag{4.7}$$

En la anterior expresión analítica, se identifican los siguientes parámetros que caracterizan el modelo de rendimiento:

- m_o: latencia de la red, en milisegundos (ms).
- λ: coste de comunicación por byte (inverso del ancho de banda), en ms/byte.
- V: volumen total de comunicación, en bytes.
- α: porción de V enviado a los workers.
- T_c: tiempo total de procesamiento, en ms.
- n: número de workers actual de la aplicación.

En conclusión, el índice de rendimiento Pi permitirá obtener para una aplicación Master/Worker el número de workers que maximiza el rendimiento, minimizando tiempo de ejecución, sin desperdiciar recursos, independientemente del valor de los parámetros que caracterizan a dicha aplicación.

4.4 Definición del modelo de rendimiento para sintonización dinámica

Los problemas de rendimiento que resuelven las dos estrategias planteadas en las secciones anteriores se caracterizan porque dependen de condiciones dinámicas, tales como la cantidad de tareas disponibles o la carga de los procesadores; de modo que, dichos problemas son apropiados para ser resueltos dinámicamente. Por tanto, la integración en MATE del modelo de rendimiento que contiene las dos estrategias descritas, permitirá, mediante el proceso de sintonización dinámica, intentar resolver esos problemas partiendo de la situación más adecuada.

Dicha integración se ha realizado mediante el diseño e implementación del tunlet adecuado. Como se comentó en la sección 3.3.3 del capítulo 3, los tunlets son el núcleo de la sintonización automática y dinámica implementada por MATE, en términos de representación del conocimiento y constituyen el mecanismo inteligente empleado por MATE en la fase de análisis. Cada tunlet define e implementa una particular técnica de sintonización, de tal modo que en nuestro caso de estudio, el tunlet que se ha diseñado plantea la lógica de análisis necesaria para aplicar el modelo de rendimiento estudiado para aplicaciones Master/Worker.

Es de destacar que el modelo de rendimiento planteado en las secciones previas fue desarrollado en el mismo grupo de investigación que ha diseñado e implementado el entorno de sintonización dinámica MATE y se encuentra estrechamente asociado con dicha herramienta. Por eso, la definición y estructura del modelo de rendimiento se adapta a la organización del conocimiento requerida por MATE en la fase de análisis durante el proceso de sintonización. Es decir, en las estrategias definidas se pueden distinguir (a) el conjunto de puntos de medida que deben ser monitorizados, (b) las expresiones de rendimiento a evaluar con dichos puntos de medida y (c) los parámetros críticos a modificar para mejorar el rendimiento de la aplicación

En el caso de la estrategia de balanceo de carga detallada anteriormente, MATE como entorno de sintonización dinámica, para poder modificar de forma automática y dinámica el factor de distribución solo necesita monitorizar el tiempo (tc_i) que emplea cada worker en el procesamiento del chunk asignado para poder estimar de este manera la media μ_C y la desviación estándar σ_C. De este modo, en la tabla 4.1 se presenta la definición formal de esta primera estrategia del modelo de rendimiento.

Puntos de Medida	- tc_i: tiempo de cómputo de cada worker, en ms.
Expresiones de rendimiento	Expresión analítica para obtener el factor de distribución del primer batch de la iteración: $x_0 = 1 + \left(\sigma_C\sqrt{N/2}\right)/\mu_C$ Expresión analítica para obtener el factor de distribución del resto de los batches a lo largo de la iteración $x_j = 2 + \left(\sigma_C\sqrt{N/2}\right)/\mu_C$
Puntos/Acciones de sintonización	El factor de distribución será el elemento a sintonizar. Su valor puede ser modificado a lo largo de toda la iteración.

Tabla 4.1 Definición de la estrategia de balanceo de carga para su uso bajo sintonización dinámica

Por otro lado, para determinar el número de workers que maximiza el rendimiento,

MATE necesita monitorizar la latencia de la red (m_0) y el coste de comunicación por byte (λ) como parámetros que caracterizan el sistema de cómputo. Además para calcular el volumen total de comunicación *V* que tiene lugar en la aplicación, es necesario determinar el tamaño de las comunicaciones establecidas entre el master y los workers (v_i y v_m). Y finalmente el tiempo de cómputo total T_c será calculado mediante el tiempo de procesamiento (tc_i) de cada uno de los workers que participan en la ejecución. La siguiente tabla 4.2 muestra la definición formal de la segunda estrategia que compone el modelo de rendimiento estudiado.

Puntos de Medida	- m_0: latencia de la red, en ms. - λ: coste comunicación por byte, en ms/byte. - v_i: tamaño de las tareas enviadas al worker *i*, en bytes. - v_m: tamaño de los resultados enviados al master desde cada worker, en bytes. - tc_i: tiempo de cómputo de cada worker, en ms
Expresiones de rendimiento	La expresión que se ha de evaluar es el índice de rendimiento para distinto número de workers: $$Pi(x) = \frac{xT_t(x)^2}{T_c}$$ Donde el tiempo de ejecución de una iteración $T_t(x)$ es $$T_t(x) = 2m_o + \frac{\left[\left((x-1)\alpha + 1\right)\lambda V + T_c\right]}{x}$$
Puntos/Acciones de sintonización	El número de workers de la aplicación será el elemento a sintonizar. El nuevo valor será aquel que prediga un menor tiempo de ejecución y mejor aprovechamiento de los recursos.

Tabla 4.2 Definición de la estrategia de determinación del número de workers para su uso bajo sintonización dinámica

Capítulo 5

Desarrollo del modelo de rendimiento en MATE

5.1 Introducción

En el trabajo de investigación expuesto hasta el momento, se ha presentado la metodología de investigación seguida para conseguir el objetivo establecido inicialmente: *sintonizar mediante MATE una aplicación empleada en computación de altas prestaciones desarrollada bajo un paradigma Master/Worker*.

Inicialmente se comenzó por un estudio en profundidad de la herramienta MATE. Recordando lo expuesto en el capítulo 3, MATE (*Monitoring, Analysis and Tuning Environment*) es, como su nombre indica, una herramienta creada para adaptar y controlar la ejecución de aplicaciones paralelas. Este entorno, trabaja sobre la aplicación en tres fases diferentes y continuas: monitorización, análisis y sintonización. Inicialmente instrumenta una aplicación durante el tiempo de ejecución de forma dinámica y automática para obtener información sobre el comportamiento de dicha aplicación. La fase de análisis busca los problemas, detecta sus causas y proporciona las soluciones para eliminar esos problemas de rendimiento. Finalmente, MATE sintoniza la aplicación aplicando las soluciones dinámicamente.

Para que la fase de análisis tenga lugar, MATE necesita poseer el conocimiento sobre el/los problemas de rendimiento que se quieren resolver. Los modelos de rendimiento constituyen dicho conocimiento empleado por MATE para conducir el proceso de análisis, determinando la información que se necesita recopilar durante la ejecución (*puntos de medida*), como evaluar la información recogida (*funciones de rendimiento*) y que cambios se necesitan para sintonizar la aplicación (*puntos/acciones/sincronizaciones de sintonización*).

De tal modo, el segundo paso realizado en nuestra metodología ha sido el estudio de un modelo de rendimiento para aplicaciones Master/Worker, cuyo principal objetivo es resolver los problemas de rendimiento que en ellas se presentan: desbalanceo de carga entre los workers y empleo del adecuado número de workers. En la sección 4.4 del capítulo 4, se presenta la representación de dicho modelo en los términos los puntos de medida, funciones de rendimiento y puntos de sintonización.

Por tanto, una vez conocida la herramienta de sintonización y estudiados los problemas de rendimiento que se desean resolver y su representación, el último paso para lograr el objetivo del presente trabajo, es el diseño y desarrollo del tunlet para ser integrado en MATE.

Los tunlets constituyen el núcleo de la sintonización dinámica y automática implementada por MATE, en términos de representación del conocimiento. Técnicamente, un tunlet es una librería que condensa la información sobre un determinado problema de rendimiento que puede afectar a un tipo de aplicaciones paralelas, implementando una particular técnica de sintonización. En el presente trabajo, el tunlet implementado encapsulará toda la información necesaria derivada del modelo de rendimiento estudiado.

El conocimiento presente en el tunlet será usado para las fases de monitorización, análisis y sintonización a lo largo de la ejecución de la aplicación bajo MATE. Para ser capaz de cooperar con MATE, la implementación del tunlets estará basada en la API de Sintonización Dinámica proporcionada por el módulo de análisis de la herramienta.

Para poder aplicar el modelo de rendimiento estudiado para aplicaciones Master/Worker, encapsularlo en un tunlet e integrarlo en MATE, se realizó una compleja búsqueda de aplicaciones situadas bajo dicho paradigma y empleadas en computación paralela/distribuida. Esta búsqueda permitió llegar a la conclusión de que actualmente las aplicaciones Master/Worker no presentan un uso muy extendido en computación de altas prestaciones debido al cuello de botella que supone la comunicación establecida entre un único master y todos los workers.

A pesar de ello, y con el fin de poder obtener los conocimientos deseados en el presente trabajo de investigación, se optó por una aplicación paralela/distribuida desarrollada en el

Departamento de Arquitectura de Computadores y Sistemas Operativos de la Universidad Autónoma de Barcelona. Se trata de un simulador de incendios forestales, denominado Xfire [10].

En el presente capítulo, se presenta el desarrollo del modelo de rendimiento en MATE. En la siguiente sección se presenta detalladamente la aplicación que va ser objeto de la sintonización. Seguidamente se plantea la metodología seguida en el diseño y desarrollo del tunlet, la cual se concreta con la interpretación de las dos técnicas de mejora de rendimiento de aplicaciones Master/Worker bajo dicha metodología. Finalmente se presentan las pruebas experimentales realizadas y los resultados obtenidos.

5.2 Xfire

5.2.1 Simuladores de incendios forestales

Los fuegos forestales son uno de los mayores riesgos medioambientales, especialmente en el sur de Europa. El diagnóstico de la variabilidad y propagación espacial de los mismos en un territorio requiere de la disponibilidad de una base científico-técnica, desde la cual se pueda ayudar y/o sustentar la toma de decisiones. La disponibilidad de aplicaciones informáticas en los que se integran el conjunto de variables que identifican la propagación y emisión energética de las llamas, constituye un elemento de apoyo para las estrategias de defensa de la superficie forestal ante grandes incendios forestales.

En los últimos años, los simuladores de la propagación de los incendios forestales se han afincado como un instrumento más para la toma de decisiones de los gestores forestales, útiles para decidir qué acciones son las más adecuadas para minimizar los riesgos o daños de un fuego.

La simulación de la propagación de los incendios forestales mediante las aplicaciones informáticas se fundamenta en la modelización de combustibles y en las fórmulas semi-empíricas desarrolladas por Rothermel [49]. El modelo de Rothermel es uno de los modelos más utilizados para predicción del comportamiento del fuego. La mayoría de los simuladores de comportamiento del fuego basan sus cálculos en este modelo. Sus operaciones calculan el índice de máxima propagación y la intensidad de reacción del fuego conociendo ciertas propiedades del combustible y del ambiente donde se desarrolla el incendio.

En la literatura existen varios modelos de propagación de fuegos forestales [50] los cuales parten de la idea de que la propagación de un fuego es un problema muy complejo que involucra varios aspectos que deben ser considerados relacionados con características meteorológicas (viento, temperatura, etc), de vegetación y topografía.

En el presente trabajo se ha seleccionado para ser objeto de sintonización dinámica la aplicación de simulación de la propagación de incendios forestales Xfire. Esta aplicación parte su análisis de la geometría actual del frente del incendio, para evaluar su posible avance considerando los diferentes aspectos climáticos, vegetación y topografía del terreno. En el siguiente apartado se expone con mayor detalle las principales características de este simulador.

5.2.2 Visión general de Xfire

Xfire [10] [51] es una aplicación paralela de cómputo intensivo que simula la propagación de incendios forestales.

Xfire simula la propagación de la línea de fuego basándose en el modelo de Andre-Viegas [52] [53], cuyo ciclo de operación se muestra en la figura 5.1. Xfire define la línea de fuego como un conjunto de secciones donde cada sección contiene un conjunto de puntos. Cada sección debe ser desglosada para calcular el progreso individual de cada punto en cada paso de tiempo. Cuando el progreso de todos los puntos ha sido calculado, es necesario agregar las nuevas posiciones de los puntos para reconstruir la línea de fuego.

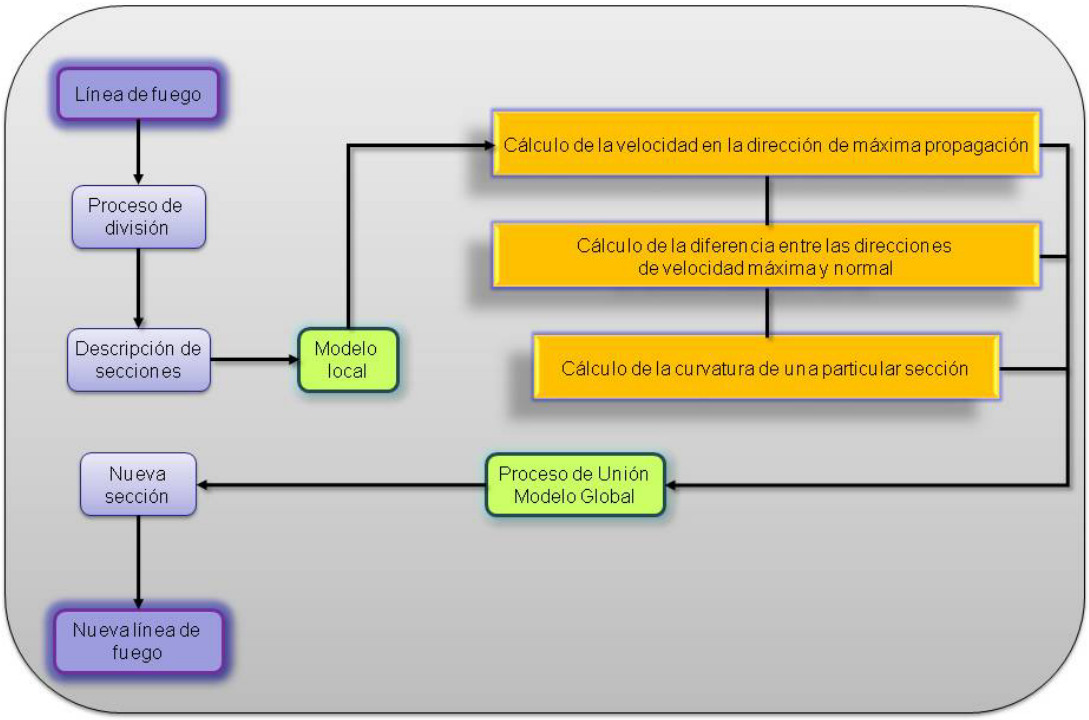

Figura 5.1 Ciclo de operación del modelo de André-Viegas

Para simular la propagación de la línea de fuego, Xfire divide el frente de fuego empleando dos modelos: uno global y otro local. El modelo global permite la partición de la línea de fuego en secciones y la unión de estas secciones en la siguiente posición del frente aplicando algoritmos numéricos. Mientras se calcula una nueva posición de la línea de fuego, el frente del fuego puede expandirse y bajo determinadas circunstancias se pueden añadir más

puntos. Se debe tener en cuenta que las secciones son independientes, pero los puntos finales de cada sección son compartidos entre secciones vecinas. El modelo local calcula el movimiento de cada punto individual. Mientras se evalúa un punto, se usan algoritmos numéricos y se tiene en cuenta condiciones estáticas y dinámicas definidas en el modelo numérico (características meteorológicas, vegetación y topografía).

La simulación de la propagación de incendios forestales implica diferentes pasos que requieren complejos cálculos que hacen que el proceso sea costoso computacionalmente en lo referente al tiempo consumido. La primera implementación de proyecto Xfire fue secuencial y se ejecutó en un PC. Sin embargo, debido al pobre rendimiento, los desarrolladores de Xfire decidieron implementarla de forma paralela. Para ello emplearon paralelismo de datos, mediante el cual el movimiento de cada sección que compone la línea de fuego puede ser calculado independientemente. De esta forma la línea de fuego se divide en N secciones, y cada sección es ejecutada por distintos procesos que forman parte de la máquina paralela.

Inicialmente Xfire se implementó empleando como librería de comunicación paralela PVM. Posteriormente, y debido al avance en el uso de la librería de paso de mensajes MPI, también fue implementada bajo dicha librería. Ambas implementaciones seguían un paradigma de implementación paralela Master/Worker. El algoritmo general de esta aplicación usando un paradigma Master/Worker es el siguiente:

- Proceso master
  - Obtienen la línea de fuego inicial.
  - Genera una partición de la línea de fuego y la distribuye entre los workers.
  - Espera la respuesta de los workers.
  - Si el tiempo de simulación ha terminado entonces finaliza la ejecución, en caso contrario, compone una nueva línea de fuego, añadiendo puntos si es necesario y vuelve al segundo paso.

- Proceso Worker
  - Obtiene la sección de línea de fuego enviada por el master.
  - Calcula la propagación local de cada punto de la sección (para calcular la posición de un punto del modelo necesita conocer sus vecinos).
  - Devuelve la nueva sección al master.

Tal y como se muestra, el proceso de simulación de fuegos realizado por Xfire es iterativo, de manera que en el algoritmo se realizan tantas iteraciones como el tiempo de simulación establecido permita.

Como datos de entrada, Xfire precisa de información que describa el terreno y las condiciones en las que se produce el incendio y la línea de fuego inicial. Para ello, durante el proceso de inicialización del algoritmo, Xfire toma los siguientes datos de entrada:

- Frente o línea de fuego inicial.

  Se caracteriza por la forma que puede presentar el frente (punto, línea, curva abierta o curva cerrada), el número de puntos que componen dicho frente y las coordenadas UMT de los mismos.

- Características meteorológicas.

  Se trata básicamente de información sobre la situación del viento. Se proporciona las coordenadas UMT del viento, su dirección en grados y su velocidad en kilómetros por hora.

- Vegetación.

  Se determina el tipo de modelo de vegetación que se encuentra sobre el terreno donde se produce el incendio y los distintos parámetros que lo caracterizan, los cuales serán utilizados en el modelo de simulación que emplea Xfire.

- Topografía.

  El terreno donde tiene lugar el incendio queda determinado por el conjunto de puntos que caracterizan el modelo de vegetación. Estos puntos quedan establecidos por coordenadas x, y, z y c.

- Tiempo de simulación.

  Para especificar el tiempo de simulación, la aplicación necesita tomar como datos de entrada el tiempo inicial de simulación, el tiempo final y el incremento de tiempo a simular en cada iteración del proceso.

5.2.3 Adaptación de Xfire al modelo de rendimiento

Como se comentó anteriormente, Xfire sigue un paradigma Master/Worker que explota el paralelismo de datos presente en la funcionalidad del simulador de incendios. Para ello, el master distribuye la línea de fuego entre todos los workers sin emplear ninguna técnica de balanceo de carga; de tal modo que en cada iteración el frente de fuego es dividido en secciones de igual tamaño que serán procesadas por los workers.

En nuestro trabajo de investigación, se ha estudiado y adaptado Xfire para poder sintonizarla siguiendo el conocimiento proporcionado por el modelo de rendimiento Master/Worker expuesto en el capítulo 5, el cual intenta resolver los problemas de rendimiento relacionados con el desbalanceo de la carga entre los workers y el uso de un apropiado número

de workers en la aplicación. Para ello ha sido necesario realizar un análisis del código fuente de la misma; en concreto nuestro estudio se ha centrado en la sección de código del proceso master encargada de la distribución del frente de fuego entre los workers. La adaptación realizada queda caracteriza por dos aspectos principales:

- Transformación de la lógica seguida en el proceso de distribución de trabajo de forma que permita la gestión de la línea de fuego como un conjunto de batchs de tamaño variable que se irán creando a lo largo de la iteración.

- La determinación del tamaño de los batchs, tal y como se comentó en la sección X, varía a lo largo de una iteración de la simulación dependiendo de un factor de partición cuyo valor refleja las condiciones actuales de balanceo de carga de la aplicación. Por tanto, es necesaria la introducción de dicho factor como variable que forma parte de la lógica de reparto del frente de fuego entre workers y que será sintonizada en el proceso de mejora de rendimiento seguido por MATE.

Es de destacar el hecho de que si la aplicación Xfire hubiese sido creada empleando un framework conocido de diseño y programación paralela, se reduciría la complejidad que alcanza el proceso de sintonización, ya que se conocería de antemano el paradigma de programación de la aplicación y su estructura funcional.

5.3 Metodología de diseño y desarrollo del tunlet

5.3.1 Metodología

El objetivo que se persigue en este capítulo es encapsular en un tunlet toda la información de cómo resolver los problemas de rendimiento definidos en el modelo estudiado. Este tunlet será usado a lo largo de la ejecución de Xfire para dirigir su monitorización, análisis y sintonización a través de la herramienta MATE.

Para realizar un correcto diseño y desarrollo del tunlet, hay que tener en cuenta que su definición debe incluir la identificación e interpretación de una serie de elementos vinculados principalmente con el modelo de rendimiento y la aplicación bajo estudio. La interdependencia entre estos componente se refleja en la figura 5.2.

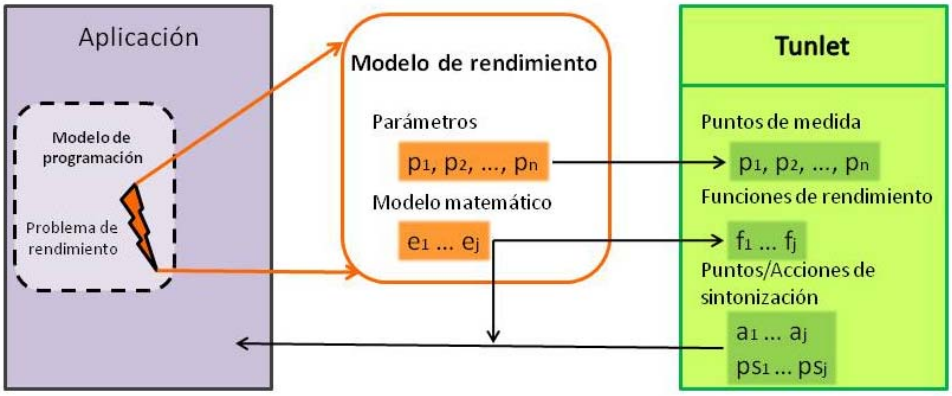

Figura 5.2 Interrelación entre la aplicación, el modelo de rendimiento y el tunlet

En lo referente al modelo de rendimiento para aplicaciones Master/Worker en nuestro caso de estudio, constituye la base del tunlet que se ha definido, debido a que proporciona información relativa a qué debería ser medido en la aplicación, cómo detectar y resolver los posibles problemas de rendimiento y qué es necesario modificar para ello en dicha aplicación. Entonces desde el punto de vista del modelo de rendimiento es necesario definir:

- Los puntos de medida.
- Las funciones de rendimiento analíticas.
- Puntos/Acciones/Sincronizaciones de sintonización.

Sin embargo, para instrumentar la aplicación en los procesos de monitorización y sintonización, el modelo de rendimiento no es suficiente, se necesita algún conocimiento adicional sobre la aplicación, tal como las variables las cuales serán usadas como métricas, los valores que van a ser cambiados, y el modelo de programación, entre otros, para tener una visión conceptual de la aplicación. Por tanto, desde el punto de vista de la aplicación, en nuestro trabajo hemos tenido en cuenta:

- El modelo de programación que sigue la aplicación, identificando los diferentes tipos de procesos involucrados en el esquema.
- Las variables o valores que se puede manipular, con el objetivo de localizar las variables a sintonizar.
- Las funciones cuya ejecución necesitamos detectar para recopilar información sobre el comportamiento de la aplicación y enviarla como eventos.

Con el propósito de poder modelar correctamente todas las interrelaciones existentes entre la aplicación Xfire, el modelo de rendimiento para aplicaciones Master/Worker y el mecanismo de encapsular el conocimiento en MATE, en el presente trabajo de investigación se ha seguido una metodología de diseño y desarrollo del tunlet. El proceso que ésta describe estimamos que es el más adecuado para culminar con un tunlet que guie el proceso de

sintonización deseado, de tal manera que puede ser generalizable para la sintonización bajo MATE de cualquier aplicación empleando cualquier modelo de rendimiento adecuado a ésta.

La metodología definida consta de los siguientes pasos:

- **Proporcionar un modelo de rendimiento.**

  En nuestro trabajo de investigación se trata del modelo de rendimiento detallado en el capítulo 4. Tal y como se expuso, se trata de un modelo preexistente desarrollado en anteriores investigaciones para aplicaciones bajo paradigma Master/Worker.

- **Comprensión del modelo de rendimiento.**

  Una vez que el modelo de rendimiento se ha determinado, la comprensión del mismo es un requerimiento básico debido a que el modelo debe de ser interpretado en concordancia con la aplicación, en este caso Xfire.

  En nuestro caso, el proceso de comprensión del modelo ha permitido establecer la correspondiente relación entre dicho modelo y el tunlet a desarrollar, mediante la caracterización de los parámetros y las funciones de rendimiento del mismo.

- **Interpretación del modelo de rendimiento.**

  Este es la fase que conlleva establecer que valores, variables y funciones de la aplicación se emplearán para interpretar los parámetros del modelo de rendimiento, definir los eventos que se deben capturar, la información asociada a los mismos y los distintos procesos que participan en la ejecución de la aplicación.

Como se puede observar, las dos primeras fases de la metodología han sido ya desarrolladas y expuestas en el capitulo precedente. De modo que el procedimiento de interpretación del modelo de rendimiento es el proceso requerido para completar el diseño y desarrollo del tunlet. Esta es la fase que se pretende mostrar en las siguientes secciones.

5.3.1.1 Interpretación del modelo de rendimiento

Cuando el modelo de rendimiento ha sido determinado y especialmente los distintos parámetros de rendimiento se han comprendido, la fase de interpretación del modelo es fundamental ya que en ella se identifican qué entidades de la aplicación se corresponden con los parámetros de rendimiento.

Este proceso incluye una serie de pasos relacionados con el estudio de la correspondencia entre los distintos componentes del modelo de rendimiento y la estructura y variables de la aplicación. Estos pasos son los siguientes:

- **Identificación de los actores en la aplicación.**

  En general, cada aplicación paralela posee distintos tipos de procesos ejecutándose en paralelo y cooperando para resolver el problema. De modo que cada tipo de proceso en el modelo de programación representa un actor.

  El tunlet que se diseñe necesita poseer esta información porque en general cada tipo de proceso debe ser instrumentado de forma diferente dependiendo del papel que juega en la ejecución de la aplicación. Es decir, para cada actor, se necesitan capturar eventos distintos, de tal modo que la instrumentación insertada en los mismos también variará.

- **Identificación de la información/variables/valores.**

  Los parámetros de rendimiento que caracterizan el modelo tienen que ser interpretados de acuerdo a las variables, valores y funciones de la aplicación bajo estudio, en este caso Xfire.

  El concepto de variable, corresponde a variables presentes en la aplicación; mientras que un valor es el contenido de un determinado parámetro de una función o el resultado de la misma. Tanto las variables como los valores pueden ser requeridos para cambiar su valor o bien obtenerlo, es decir, son los elementos de la aplicación que pueden ser instrumentados o sintonizados.

  En función de las características o naturaleza de cada parámetro de rendimiento, en nuestro proceso de diseño debemos determinar cómo constituir su valor y cuál es el evento que se debe definir para obtener la información asociada.

  En el desarrollo del tunlet el empleo de variables requiere una especial importancia, ya que aquellas variables de las que se decida que es necesario requerir su valor o cambiarlas, tienen que ser variables globales, condición expuesta por la librería Dyninst al utilizarla como método de instrumentación dinámica. De tal modo, puede ocurrir que se necesitan determinados cambios en la implementación de la aplicación definiendo algunas variables como globales.

- **Identificación de los eventos.**

  Los eventos constituyen el mecanismo empleado por MATE para recopilar información sobre el comportamiento de la aplicación. Los eventos son capturados en las entradas o salidas de funciones y pueden contener información adicional asociada a ellos.

  Entonces, de acuerdo a la semántica de los distintos parámetros que componen nuestro modelo de rendimiento, debemos determinar cuáles son las entradas y salidas

de las funciones que tienen que ser capturadas, es decir, localizar aquellos puntos de nuestra aplicación en los cuales se tiene disponible la información que se requiere.

- **Determinación de cuándo y bajo qué circunstancias evaluar las funciones de rendimiento y ejecutar el cambio de los puntos de sintonización.**

  La recopilación de los eventos no solo permite al tunlet ir recopilando la información necesaria para evaluar las funciones de rendimiento asociadas a la estrategia de sintonización, sino que también determina cuando se debe evaluar dichas condiciones y, bajo determinadas condiciones, ejecutar el cambio de los parámetros críticos que permitirán la mejora de rendimiento de la aplicación.

Esta metodología será la empleada para la interpretación de las dos técnicas de sintonización estudiadas para mejorar el rendimiento de aplicaciones Master/Worker.

5.3.1.2 Requerimientos de MATE

Las fases de la metodología de diseño del tunlet expuestas en la sección 5.3.1, presentan aspectos que son muy dependientes de los detalles de implementación de MATE, ya que se encuentran muy relacionados con la manera en la que el Analizador representa y usa el conocimiento.

Si recordamos lo expuesto en el capitulo X (MATE), desde un punto de vista funcional, el Analizador está dividido en dos partes principales que son la API de Sintonización Dinámica (DTAPI) y los tunlets.

La DTAPI constituye la interfaz que emplea el Analizador para comunicarse con las fases de Monitorización y Sintonización. Esta API proporciona al Analizador una visión global de la aplicación, las tareas y los eventos. Con respecto a los tunlets, ellos usan la API de Sintonización Dinámica para gestionar la aplicación e invocar las peticiones de monitorización y sintonización, y para manejar los eventos recopilando la información de la aplicación necesaria para el análisis. Por tanto, la DTAPI constituye la interfaz que el tunlet debe emplear para trabajar de forma coordina con MATE.

Para la implementación del tunlet, en el estudio realizado se ha considerado tener presente además los siguientes aspectos:

- *Cómo capturar la información.* Se han considerado los métodos proporcionados por la DTAPI para instrumentar la aplicación, y en particular, para determinar las propiedades que definen un evento. Hay que tener presente que para insertar un evento en un

proceso particular, se necesita un nombre para identificar el evento, donde se debe insertar dicho evento en el código fuente y los atributos asociados a él.

- *Como gestionar la información recopilada.* Cuando un evento es insertado en un proceso, es necesario determinar una entidad que controle ese evento cuando éste ocurre y es recibido en la fase de análisis. Generalmente, los tunlets son los que controlan los eventos, debido a que en ellos se encapsula la lógica para procesar la información e interpretarla de acuerdo al modelo de rendimiento.
- *Como y donde manejar la información.* En el presente trabajo el tunlet desarrollado gestiona una compleja estructura de datos para cada iteración de la aplicación. En tal estructura, la información recopilada es almacenada en función de su procedencia: información sobre la iteración, sobre los batches repartidos en la iteración y sobre los workers.
- *Cómo modificar la aplicación.* De forma similar a la instrumentación para la monitorización, cuando se requiere alguna información de sintonización se han de considerar los métodos proporcionados por la DTAPI para introducir algunas modificaciones en la aplicación. En nuestro trabajo, la modificación de la aplicación únicamente se realizará mediante el cambio del valor de alguna variable.

Teniendo en cuenta los conceptos expuestos en la fase de interpretación del modelo de rendimiento y su dependencia con la implementación, en las próximas secciones se exponen la especificación del tunlet creado a partir del modelo de rendimiento expuesto en el capitulo X y la aplicación Master/Worker Xfire.

5.4 Desarrollo del tunlet

En la presente sección se muestra la especificación del tunlet que integrará las dos estrategias de sintonización para mejorar el rendimiento en aplicaciones Master/Worker:

- Balanceo de la carga entre los workers.
- Adaptación del número de workers.

La especificación del tunlet se realizará teniendo en cuanta la especificación de cada una de las estrategias de sintonización, las cuales fueron expuestas detalladamente en el capítulo 4. Además la generación del tunlet se detallará partiendo de la metodología diseñada y presentada en la sección 5.3.1.1.

En la presente sección se aplica la metodología detallada en la sección previa para definir y diseñar cada una de las técnicas de sintonización que componen el tunlet desarrollado,

teniendo en cuenta las interrelaciones entre el modelo de rendimiento, la aplicación Xfire y la herramienta de sintonización MATE.

5.4.1 Balanceo de la carga entre workers

El desbalanceo de carga en sistemas paralelos puede ser causado por la heterogeneidad de los procesadores, interferencias con el sistema operativo o irregularidad en las tareas asignadas a los procesadores.

El balanceo de carga dinámico es una técnica cuyo propósito es distribuir la carga entre los procesos para evitar que algunos procesos se mantengan ociosos mientras que otros esperan recibir trabajo y no hacen nada más. Esto se consigue asignando más trabajo a los procesadores que terminan antes el trabajo asignado.

Factoring es una estrategia de balanceo de carga que divide el número total de tareas a procesar en bacthes. Cada batch tiene tantos chunks como procesadores están ejecutándose y cada chunk contiene la misma cantidad de tareas.

Tal y como se expuso en la sección X, en el presente trabajo de investigación se ha estudiado una aproximación de la técnica de *Factoring* denominada *Dynamic Adjusting Factoring* (*DAF*). Esta estrategia permite ajustar dinámicamente el tamaño de los batches a lo largo de la ejecución teniendo en cuenta las condiciones actuales de balanceo de carga de la aplicación.

5.4.1.1 Interpretación de la técnica de sintonización

Recopilando la información detallada en la sección X, los términos y conceptos que forman parte de la técnica de sintonización para balancear la carga entre los workers son:

- N: número de workers.
- x_i: inverso del factor de partición, porción de tareas incluidas en el batch i.
- C: tiempo de procesamiento medio por tarea (*ms/tarea*).
- $\mu(C)$: media del tiempo de procesamiento por tarea.
- $\sigma(C)$: desviación estándar del tiempo de procesamiento por tarea.
- *Batch*: cada una de las partes en las que M es dividida a lo largo de la implementación mediante el algoritmo *DAF*. Cada batch constituye un subconjunto de tareas.

El objetivo de esta técnica es ajustar a lo largo de la ejecución de la aplicación el ***factor de partición*** que determina el tamaño de los distintos batches en los que se va dividiendo el conjunto total de tareas. Por tanto, dicho *factor de partición* constituye el ***punto de sintonización*** de esta estrategia.

Tal y como se detalló en la sección 4.3.1 del capítulo 4, el valor del factor de partición se determina a partir del valor de x_i. Éste se calcula mediante las siguientes expresiones:

$$x_i = \begin{cases} 1 + \left(\sigma_C \sqrt{N/2}\right)/\mu_C \quad , & si\ i = 0 \\ 2 + \left(\sigma_C \sqrt{N/2}\right)/\mu_C \quad , & si\ i \neq 0 \end{cases}$$

Ambas fórmulas analíticas constituyen las ***funciones de rendimiento*** de la técnica de sintonización. La diferencia entre ambas expresiones radica en el hecho de que en el caso de la creación del primer batch (cuando $i = 0$) los workers están sincronizados ya que se encuentran el inicio de la iteración en espera de datos para ser procesados, mientras que para el resto de los batches la disponibilidad de los workers depende de la velocidad con la que los previos batches fueron procesados.

Así, para el cálculo de x_i necesitamos obtener el tiempo medio de procesamiento por tarea (C), el cual es necesario para calcular μ_C y σ_C, y el número de workers N de la aplicación.

Identificación de actores.

La aplicación que va a ser sintonizada, tal y como se expuso en secciones previas, está implementada bajo un paradigma Master/Worker. De tal modo, que en su ejecución podemos identificar dos tipos de procesos que cooperan entre sí: un proceso master y N procesos worker.

Identificación de variables y valores.

Para esta técnica de sintonización, se necesitan interpretar *xi*, C y N, identificando las variables y valores que en la aplicación representan estos parámetros.

- C tiene que ser calculado como la media del tiempo de cómputo empleado por un worker en procesar cada una de las tareas recibidas. De tal modo, el parámetro puede calcularse como $C = media(tc_i/NTareas)$, donde:
    - tc_i equivale el tiempo que el worker i ha estado procesando las tareas asignadas. Este tiempo se calcula tomando los instantes en los que el Worker comienza y termina la fase de cómputo.
    - *NTareas* es una variable en el proceso master que indica cuántas tareas son enviadas a cada uno de los workers. También es empleada en el proceso de recepción que realiza el master a lo largo de toda la iteración, con el objetivo de comprobar que al final de la misma se han recibido todas las tareas que han sido enviadas.
- N es obtenido de la variable *NW* del proceso master. Esta variable permite al master controlar la cantidad de workers a lo largo de la iteración. Tiene que ser medida al

inicio de la iteración de la aplicación, y debe ser periódicamente actualizada en el caso de que el número de workers pueda cambiar a lo largo de la ejecución.

- x_i es el inverso del valor que tomará la variable que va a ser sintonizada (*F0, F1* y *F2*), es decir, el factor de distribución que determina el número de batchs. Esta variable se sitúa en el proceso master ya que éste es el encargado de realizar la distribución de las tareas entre los workers. Desde el punto de vista de la aplicación ha sido necesaria la definición de 3 variables que actúan como factores de distribución, denominadas *F0*, *F1* y *F2*. Esto se debe principalmente a la adaptación de la aplicación Xfire a la lógica del proceso de sintonización:
    - La variable F0 representa el factor de distribución para el primer batch de la iteración.
    - La variable F1 representa el factor de distribución para el segundo batch de la iteración.
    - La variable F2 representa el factor de distribución para el resto de los batches de la iteración.

Por último hay que tener en cuenta que son necesarias una serie de variables de control que permiten crear el flujo propio de la lógica de análisis que corresponde a la técnica de sintonización. Estas variables son:

- *workerId*, esta variable es usada por el proceso master para identificar el proceso worker al que está enviando o de que proceso worker está recibiendo.
- *batchId*, esta variable está presente en el proceso master. Es necesaria porque existen eventos cuya información está relacionada con un determinado batch. En concreto, está variable es usada por el proceso master bajo dos circunstancias:
    - Para identificar cada uno de los batches que se van creando a lo largo de la iteración en función del factor de partición.
    - Para identificar a que batch pertenece las tareas que está enviando a un determinado proceso worker o que recibe de un determinado proceso worker.
- *numChunk*, esta variable es empleada por el proceso master para tener conocimiento sobre el número de chunks que componen los batches que se van creando. Generalmente, los batches tienen tantos chunks como workers están implicados en la ejecución de la aplicación pero existe la excepción de los últimos batches de la iteración.
- *TheTotalWork,* esta variable empleada en el proceso master representa en la aplicación al número total de tareas de una iteración, es decir, el número de puntos totales de la actual línea de fuego.

- *iterId*, esta variable está presente en el proceso master y en el proceso Worker. Para el master esta variable se requiere para identificar cuando se recibe un evento a que iteración pertenece para asociar la información a una correspondiente iteración. De manera semejante, en el caso del proceso Worker, la identificación de la iteración es necesaria para asociar a una iteración determinada la información de los eventos generados en dichos procesos.

Identificación de eventos

El siguiente paso consiste en determinar cómo, cuándo y dónde capturar las variables y valores previamente enumerados. Para ello se han definido una serie de eventos que serán insertados en los procesos correspondientes y proporcionarán la información requerida por la técnica de sintonización. Los eventos definidos son los siguientes:

- Inicio de iteración.

  Este evento se genera cada vez que el proceso master comienza una nueva iteración en la aplicación, es decir, cuando comienza la distribución de la nueva línea de fuego entre los workers.

  La información que presenta asociada es el número de iteración en la que se encuentra la aplicación, el número de tareas totales a distribuir a lo largo de esa iteración y el número de workers que participan.

  Para obtener estos datos, el evento se debe de insertar en la entrada de la función *global_sendreceive* que realiza el proceso de distribución de datos.

- Fin de iteración.

  Este evento es capturado cuando el proceso master termina la recepción de los resultados obtenidos por los workers.

  La información que presenta asociada es el número de la iteración que termina. Y de la misma forma que el evento anterior es insertado en la función *global_sendreceive*, pero en este caso a la salida de la misma.

- Envío de datos del master al worker.

  Este evento se genera cada vez que el proceso master envía el conjunto de tareas que debe computar a un proceso worker.

  Este evento permite recopilar información relacionada con el identificador del worker al que se le envía las tareas, el identificador del batch al que pertenecen dichas tareas y el número de tareas enviadas. Para ello, el evento debe ser insertado en la

entrada del método *global_sendwork*, en el cual se realiza el empaquetado y envío de todos los datos necesarios al worker determinado.

- Recepción de datos de los workers en el master.

  Este evento tiene lugar cada vez que el proceso master recibe el resultado del procesamiento realizado por un worker determinado.

  Tiene asociada la misma información que el evento anterior, pero en este caso debe ser insertado a la salida de la función *global_receivework*, en la cual se realiza la recepción y desempaquetado de la información recibida por el master.

- Inicio cómputo de worker.

  Este evento se genera cuando un worker comienza el procesamiento del conjunto de tareas recibidas.

  Mediante este evento se requiere recopilar información relacionada con la iteración en la cual se encuentra computando el worker, el identificador del batch al cual pertenece el chunk sobre el que va a computar, y la marca de tiempo asociada a dicho comienzo de cómputo.

  Para obtener correctamente dicha información, el evento debe ser insertado a la entrada de la función *arcStepKernel*, la cual funcionalmente representa el proceso de cómputo realizado por cada worker.

- Fin de cómputo de Worker.

  Este evento complementa al evento anterior, y se genera cuando un worker finaliza el cómputo del chunk enviado.

  Se recopila la misma información que en evento de inicio de cómputo, pero en la salida de la función *arcStepKernel*.

- Creación de un nuevo batch

  Este evento se genera cada vez que el proceso master durante la fase de reparto de tareas entre los workers calcula un nuevo batch con un determinado factor de distribución.

  La información asociada a este evento es el identificador del nuevo batch creado y su tamaño, es decir, el número de chunks que lo componen. Para recopilar dicha información el evento debe ser insertado en la función *Factoring_SetNumTuples,* que forma parte del procedimiento de *factoring,* en la cual se van inicializando las estructuras de datos que representan a los batches.

La tabla 5.1 resume la información más importante relacionada con cada evento.

Evento	Atributos	Actor	Método	Lugar
Inicio Iteración	- iterId - NW - TheTotalWork	master	global_sendreceive	entrada
Fin Iteración	- iterId	master	global_sendreceive	salida
EnvioMW	- workerId - batchId - NTareas	master	global_sendwork	entrada
RecepciónMW	- workerId - batchId - NTareas	master	global_receivework	salida
Inicio Cómputo Worker	- Marca de tiempo - iterId - batchId	worker	arcStepKernel	entrada
Fin Cómputo Worker	- Marca de tiempo - iterId - batchId	worker	arcStepKernel	salida
Creación de un nuevo batch	- batchId - numChunks	master	Factoring_SetNumTuples	salida

Tabla 5.1 Información sobre los eventos para la técnica de sintonización de balanceo de carga

Determinación de cuándo y bajo qué condiciones realizar el proceso de sintonización

La recepción de determinados eventos desencadena en la lógica de sintonización implementada en el tunlet la evaluación de las funciones de rendimiento propias de la estrategia de balanceo de carga.

En el caso bajo estudio, el ajuste del factor de distribución tiene lugar cuando se reciben los siguientes eventos:

- Inicio de iteración

  Cuando el proceso master comienza la ejecución de una nueva iteración, el tunlet ya posee la información sobre los tiempos de cómputo por tarea de la iteración anterior ya completada.

  Este hecho permite la evaluación de las formulas analíticas de la estrategia de balanceo de carga, de tal manera que se actualizan los valores de los factores de distribución *F0* y *F1* para determinar el tamaño de los batches *0* y *1* de la iteración, tal y como se expone en el paso 2 del algoritmo planteado en la sección 4.2.1 del capítulo 4.
- Fin de cómputo de un worker.

  Cuando un proceso worker termina el cómputo del chunk que le ha sido enviado, la información sobre el tiempo de cómputo es recopilada por el tunlet.

  En el caso de que, tras recibir la información de este chunk, se compruebe que se ha completado el procesamiento del batch al cual pertenece dicho chunk, se actualiza el histórico de información acumulada sobre los tiempos de cómputo por tarea con los datos asociados al batch finalizado; y entonces tiene lugar la actualización del valor del factor de distribución *F2* teniendo en cuenta las condiciones actuales de la aplicación en cuanto a balanceo de carga.

La modificación o sintonización de las variables *F0*, *F1* y *F2* en la aplicación puede tener lugar en cualquier momento a lo largo de las distintas iteraciones que componen la ejecución de la aplicación.

5.4.2 Adaptación del número de workers

La ejecución de una aplicación con el número de workers más apropiado no es una decisión trivial. En la mayoría de los casos las condiciones cambian durante la ejecución de la aplicación (por ejemplo en sistemas con carga compartida) y el número correcto de workers no es fijo, por lo que debe evolucionar durante la ejecución de la aplicación.

En estos casos, la determinación del número de workers debe ser sintonizado sobre la marcha en tiempo de ejecución de la aplicación. Con este objetivo, en el presente trabajo de investigación se ha estudiado una técnica de sintonización que permite mejorar el rendimiento de la aplicación ajustando el número de workers.

Para conseguir este objetivo, tal y como se expone en la sección X, se emplea un índice de rendimiento que relaciona tiempo de ejecución con eficiencia en cuanto al uso de los recursos, determinando el número de workers que proporciona un mejor rendimiento en la aplicación.

5.4.2.1 Interpretación de la técnica de sintonización

Recopilando la información detallada en la sección X, los términos y conceptos que forman parte de la técnica de sintonización para adaptar el número de workers en la aplicación son:

- m_o: latencia de la red, en milisegundos (ms).
- λ: coste de comunicación por byte (inverso del ancho de banda), en ms/byte.
- V: volumen total de comunicación, en bytes.
- α: porción de V enviado a los workers.
- T_c: tiempo total de procesamiento, en ms.
- n: número de workers actual de la aplicación.
- T_t: tiempo total de la iteración, en ms.

El objetivo de esta técnica es adaptar el ***número de workers*** de la aplicación a aquel valor que represente un mejor ajuste del rendimiento teniendo en cuenta el tiempo de ejecución y la eficiencia en el uso de los recursos. Por tanto, dicho *número de workers* constituye el ***punto de sintonización*** de esta estrategia.

Tal y como se expuso en la sección 4.3 del capítulo 4, el número de workers, *n*, es obtenido a partir de un índice de rendimiento *Pi(n)* expresado mediante la siguiente expresión analítica:

$$Pi(n) = \frac{nT_t(n)^2}{T_c} \quad \text{donde } T_t \text{ es}$$

$$T_t(n) = 2m_o + \frac{\left[\left((n-1)\alpha + 1\right)\lambda V + T_c\right]}{n}$$

Ambas fórmulas analíticas constituyen las ***funciones de rendimiento*** de la técnica de sintonización. De modo que el número de workers que buscamos será aquel que maximice el rendimiento sin desperdiciar recursos en la aplicación.

Así, para el cálculo de *n* necesitamos obtener el volumen total de comunicación (V), el tiempo total de procesamiento (T_c), la porción del volumen total de comunicación enviado a los workers (α), la latencia de la red (m_0), el coste de comunicación por byte (λ) y el número actual de workers (n).

Identificación de actores.

De nuevo la aplicación que va a ser sintonizada bajo esta estrategia es Xfire. Coincidiendo con lo expuesto para la especificación de la estrategia de balanceo de carga, en la ejecución de la aplicación se pueden identificar dos tipos de procesos que cooperan entre sí: un proceso master y *N* procesos worker.

Identificación de variables y valores.

Para esta técnica de sintonización, se necesitan interpretar V, T_c, m_0, λ y n identificando las variables y valores que en la aplicación representan estos parámetros.

- V tiene que ser calculado como la suma del tamaño de las tareas enviadas desde el master a todos los worker o recibidas desde los workers. De tal modo, el parámetro puede calcularse como $V = \sum_{i=0}^{n-1}(v_i + v_m)$, donde:
    - v_i equivale al tamaño total de las tareas enviadas a cada worker *i*, en bytes. Para obtener este valor se necesita:
        - El número de tareas enviadas a cada worker.
          Este valor puede ser capturado en el proceso de envío del master a los workers, en concreto a partir de la variable *NTareas* que indica el número de tareas enviadas al worker *i*.
        - El tamaño en bytes de cada tarea.
          En la aplicación, en concreto en el proceso master, existe una variable denominada *TheWorkSizeUnitBytes* que indica el tamaño en bytes de cada tarea.
    - v_m equivale al tamaño total de tareas recibidas por el master. Este valor puede ser capturado cuando el master recibe las tareas de los workers. Y el tamaño en bytes de cada tarea se obtiene de forma equivalente a v_i.
- α puede ser calculado directamente una vez obtenido el valor vi para cada worker *i* y el volumen total de comunicación V, a partir de la siguiente expresión $\alpha = \sum_{i=0}^{n-1} v_i/V$.
- T_c tiene que ser calculado como la suma del tiempo de cómputo de cada Worker a lo largo de la iteración. De tal modo, el parámetro puede calcularse como $T_c = \sum_{i=0}^{n-1}(tc_i)$, donde:
    - tc_i equivale al tiempo de cómputo total de cada worker *i*, en ms. Este valor puede ser obtenido a partir de los tiempos de comienzo y fin de la fase de cómputo de cada uno de los workers.
- m_0 y λ deberían calcularse al principio de la iteración y deberían ser periódicamente actualizados para permitir la adaptación del sistema a las condiciones de la red. En nuestro caso de estudio se han dejado como valores constantes.
- n es el valor actual de workers de la aplicación, que se puede obtener mediante la variable *NW.* Este es el valor que será empleado para sintonizar la variable *NW.*

En el diseño de esta técnica de sintonización son de nuevo necesarias una serie de variables de control que permiten crear el flujo propio de la lógica de análisis que corresponde a la técnica de sintonización para adaptar el número de workers. Estas variables son:

- *workerId*, esta variable es usada por el proceso master para identificar el proceso worker al que está enviando o de que proceso worker está recibiendo.
- *TheWorkSizeUnitBytes,* esta variable empleada en el proceso master representa cual es el tamaño en bytes de cada una de las tareas a ser procesadas.
- *iterId*, esta variable está presente en el proceso master y en el proceso Worker. Para el master esta variable se requiere para identificar cuando se recibe un evento a que iteración pertenece para asociar la información a una correspondiente iteración. De manera semejante, en el caso del proceso Worker, la identificación de la iteración es necesaria para asociar a una iteración determinada la información de los eventos generados en dichos procesos.

Identificación de eventos

De nuevo, una vez determinadas las variables y valores requeridos de la aplicación, se deben definir los eventos que permitan obtener la información que represente los datos requeridos. Los eventos definidos son los siguientes:

- Inicio de iteración.

  Este evento se genera cada vez que el proceso master comienza una nueva iteración en la aplicación, es decir, cuando comienza la distribución de la nueva línea de fuego entre los workers.

  La información que presenta asociada es el número de iteración en la que se encuentra la aplicación, el tamaño en bytes de las tareas que se van a distribuir a lo largo de esa iteración y el número de workers que participan.

  Para obtener estos datos, el evento se debe de insertar en la entrada de la función *global_sendreceive* que realiza el proceso de distribución de datos.

- Fin de iteración.

  Este evento es coincidente con el detallado en la anterior técnica de sintonización.

- Envío de datos del master al worker.

  Este evento se genera cada vez que el proceso master envía el conjunto de tareas que debe computar a un proceso worker.

  Este evento permite recopilar información relacionada con el identificador del worker al que se le envía las tareas y el número de tareas enviadas. Para ello, el evento debe ser insertado en la entrada del método *global_sendwork*, en el cual se realiza el empaquetado y envío de todos los datos necesarios al Worker determinado.

- Recepción de datos de los workers en el master.

  Este evento tiene lugar cada vez que el proceso master recibe el resultado del procesamiento realizado por un worker determinado.

  Tiene asociada la misma información que el evento anterior, pero en este caso debe ser insertado a la salida de la función *global_receivework*, en la cual se realiza la recepción y desempaquetado de la información recibida por el master.

- Inicio cómputo de worker.

  Este evento se genera cuando un worker comienza el procesamiento del conjunto de tareas recibidas.

  Mediante este evento se requiere recopilar información relacionada con la iteración en la cual se encuentra computando el Worker y la marca de tiempo asociada a dicho comienzo de cómputo.

  Para obtener correctamente dicha información, el evento debe ser insertado a la entrada de la función *arcStepKernel*, la cual funcionalmente representa el proceso de cómputo realizado por cada worker.

- Fin de cómputo de worker.

  Este evento complementa al evento anterior, y se genera cuando un worker finaliza el cómputo del chunk enviado.

  Se recopila la misma información que en evento de inicio de cómputo, pero en la salida de la función *arcStepKernel*.

La tabla 5.2 resume la información más importante relacionada con cada evento.

Determinación de cuándo y bajo qué condiciones realizar el proceso de sintonización

Bajo esta estrategia de sintonización, el ajuste del número de workers se realiza el inicio de una iteración, es decir, cuando se ha recibido un evento de inicio de iteración. La actualización del valor que corresponde al de workers tiene lugar en este instante ya que es cuando el tunlet ha podido recopilar toda la información procedente de la iteración anterior y por tanto puede obtener todos los parámetros necesarios para evaluar la función de rendimiento.

La modificación o cambio del número de workers en Xfire se realiza exclusivamente al inicio de una iteración.

Evento	Atributos	Actor	Método	Lugar
Inicio Iteración	- iterId - NW - TheWorkSizeUnitBytes	master	global_sendreceive	entrada
Fin Iteración	- iterId	master	global_sendreceive	salida
EnvioMW	- workerId - NTareas	master	global_sendwork	entrada
RecepciónMW	- workerId - NTareas	master	global_receivework	salida
Inicio Cómputo Worker	- Marca de tiempo - iterId	worker	arcStepKernel	entrada
Fin Cómputo Worker	- Marca de tiempo - iterId	worker	arcStepKernel	salida

Tabla 5.2 Información sobre los eventos para la técnica de sintonización para adaptar el número de workers

5.4.3 Integración de las estrategias de sintonización en el tunlet

Tras interpretar las dos técnicas de sintonización siguiendo la metodología desarrollada, se obtiene un tunlet que contiene el conocimiento para resolver los problemas de rendimiento presentes en aplicaciones Master/Worker.

El tunlet combina la lógica de análisis de las dos estrategias de sintonización y los eventos definidos en cada una de ellas, tal y como se muestra en la tabla 5.3.

En anteriores trabajos realizados con MATE [7] [54], se aplicaban varias técnicas de sintonización de forma separada. Durante la ejecución de la aplicación, MATE intentaba aplicar todos los escenarios de optimización, pero cada uno de manera individual. MATE cargaba todos los tunlets disponibles y cada uno de ellos llevaba a cabo su particular mejora de rendimiento de la aplicación. Su objetivo era identificar e investigar distintas técnicas de sintonización. De tal modo, que se centraban en los efectos de las técnicas individuales, sin considerar el rendimiento total de la aplicación. Pero, bajo ciertas condiciones es necesario considerar dependencias entre diferentes problemas de rendimiento y las técnicas de sintonización asociadas a ellos.

Por ello, hay que destacar que en el presente trabajo de investigación se ha desarrollado un tunlet, cuya complejidad es más elevada, ya que contiene el conocimiento necesario sobre dos técnicas de sintonización que intentan resolver los problemas de rendimiento que se observan en las aplicaciones Master/Worker ya comentados en la sección X. En este caso, la técnica de sintonización que ajusta el número de workers considera el rendimiento total de la aplicación ya que su función de rendimiento se basa en un índice que permite obtener el número de workers que no sólo minimiza el tiempo de ejecución, sino al mismo tiempo maximiza el rendimiento aprovechando de la forma más eficiente los recursos, siendo esto último también el objetivo de la estrategia de factoring implementada.

Evento	Atributos	Actor	Método	Lugar
Inicio Iteración	- iterId - NW - TheTotalWork - TheWorkSizeUnitBytes	master	global_sendreceive	entrada
Fin Iteración	- iterId	master	global_sendreceive	salida
EnvioMW	- workerId - batchId - NTareas	master	global_sendwork	entrada
RecepciónMW	- workerId - batchId - NTareas	master	global_receivework	salida
Inicio Cómputo Worker	- Marca de tiempo. - iterId - batchId	worker	arcStepKernel	entrada
Fin Cómputo Worker	- Marca de tiempo. - iterId - batchId	worker	arcStepKernel	salida
Creación de un nuevo batch	- batchId - numChunks	master	Factoring_SetNumTuples	salida

Tabla 5.3 Información sobre los eventos para las 2 técnicas de sintonización implementadas

5.5 Resultados experimentales

Una vez obtenido el tunlet que contiene el conocimiento para desarrollar el proceso de sintonización sobre aplicaciones Master/Worker y adaptada la aplicación bajo estudio Xfire, el siguiente paso consiste en validar la eficiencia y utilidad del citado tunlet cuando es integrado en MATE para realizar el proceso de sintonización dinámica.

Tal y como se detalló en las secciones previas, el tunlet desarrollado contiene la lógica de análisis de las dos estrategias de sintonización que cubren los problemas de rendimiento de aplicaciones Master/Worker: balanceo de la carga de la aplicación entre los workers y determinación del número de workers que obtiene un buen rendimiento y eficiencia en la aplicación. Sin embargo, por motivos de tiempo, la experimentación planteada sólo cubre el estudio de rendimiento obtenido mediante la aplicación de la estrategia de balanceo de carga sobre Xfire.

Los experimentos expuestos a continuación tienen como objetivo general comprobar la mejora de rendimiento en la aplicación Xfire cuando es ejecutada y sintonizada bajo la herramienta MATE empleando el tunlet desarrollado.

Las pruebas experimentales han sido llevadas a cabo en un clúster homogéneo y dedicado compuesto por 10 nodos cuya configuración se muestra en la tabla 5.4.

La configuración hardware disponible ha sido determinante a la hora de plantear nuestros experimentos. Éstos requieren una unidad de procesamiento para cada proceso worker, para el proceso master y para el componente Analizador de MATE. Por tanto, debido a la existencia de 8 nodos de cómputo, se han podido ejecutar configuraciones Master/Worker formadas por 2, 3, 4, 5, 6 y 7 workers. En el caso de los experimentos realizados con 7 workers, es de destacar que alguna unida de procesamiento ha sido compartida entre un proceso master o worker y el módulo Analizador de MATE.

Como se comentó en la sección 5.2, Xfire toma como datos de entrada distintos ficheros de configuración que describen el terreno, las condiciones en las que se produce el incendio y la línea de fuego inicial. En las pruebas experimentales realizadas, la línea de fuego está formada por 786420 puntos que forman una curva cerrada o elipse.

Se han planteado tres escenarios de ejecución de Xfire:

1. Ejecución de Xfire en su versión original.
2. Ejecución de Xfire junto con MATE pero sin aplicar la estrategia de sintonización.
3. Ejecución de Xfire junto con MATE aplicando la estrategia de sintonización.

Nodo Front-End	- Máquina clónica con placa base Asus. - Procesador Intel Pentium 4 @3.0GHz. - 1MB L2 1 GB DDR. - HD 60 GB.
Nodo File-Server	- Máquina clónica con placa base Asus. - Procesador Intel Pentium 4 @3.0GHz. - 1MB L2 1 GB DDR. - 4xHD 60 GB: el 1º para sistema y los 3 siguientes formando un RAID-5 para alojamiento de $HOME compartido por NFS de 111 GB.
8 Nodos de cómputo	- HP dc7100sff. - Procesador Intel Pentium 4 @3.0GHz. - 1MB L2 1 GB DDR. - HD 80 GB. - Tarjeta de red Broadcom NetXtreme.
Red	Toda la red interna del cluster funciona a 1 Gbps .

Tabla 5.4 Características del entorno donde se han realizado las pruebas experiementales

El planteamiento de distintos escenarios de ejecución tiene como objetivo lograr obtener conclusiones acerca de la sobrecarga introducida por MATE y la mejora de rendimiento o no generada por el proceso de sintonización. Cada experimento fue desarrollado muchas veces y se calculó la media para el tiempo de ejecución de la aplicación. La tabla 5.5 muestra los tiempos de ejecución obtenidos para cada uno de los escenarios ejecutados.

Escenario	Número de workers	2	3	4	5	6	7
1	Xfire	529,6	381,61	305,06	264,95	229,27	209,94
2	Xfire+MATE (sin sintonizar)	559,99	414,5	336,5	299,46	261,46	242,27
3	Xfire+MATE (sintonizando)	545,14	391,83	305,96	257,9	226,27	203,59

Tabla 5.5 Tiempos de ejecución de Xfire considerando distinto número de workers en los tres escenarios de ejecución presentados (en segundos).

El primer escenario muestra los tiempos de ejecución obtenidos cuando la aplicación es ejecutada en su versión original, es decir, sin la intrusión o control de la herramienta de sintonización. Los resultados presentados muestra que, para los mismos datos de entrada, el tiempo de ejecución de la aplicación disminuye al aumentar el número de workers. Se puede concluir que Xfire escala, pero las ganancias generadas no son las ideales.

El segundo escenario de ejecución muestra los resultados logrados cuando Xfire es ejecutada bajo el control de MATE, sin realizar sintonización, es decir, solo tiene lugar la inserción de la instrumentación en los distintos procesos, la generación de los eventos que son enviados al analizador, el procesamiento de dichos eventos y la evaluación las funciones de rendimiento pero no se aplica sobre Xfire el resultado de dicha evaluación.

En cuanto a la sobrecarga generada por MATE en la ejecución de Xfire, si comparamos los resultados obtenidos de los *escenarios 1* y *2*, se observa que ésta es constante independientemente del número de workers implicados en el desarrollo de la aplicación (sobrecarga absoluta) situándose en torno a los 32 segundos. Esto se debe a que cada proceso (master o worker) emiten el mismo número de eventos, y éstos son generados en paralelo.

Sin embargo hay que destacar que si la aplicación escala, sin cambiar el tamaño de los datos de entrada, cuando se aumenta el número de workers disminuyendo el tiempo de ejecución, el *overhead* que introduce MATE aumenta en proporción a dicho tiempo de ejecución de la aplicación (sobrecarga relativa), tal y como se muestra en la figura 5.3. Por tanto sería interesante estudiar la posibilidad de intentar reducir la intrusión por defecto de MATE, especialmente cuando debido al reducido tiempo de cómputo tal *overhead* deja de ser aceptable en el proceso de sintonización.

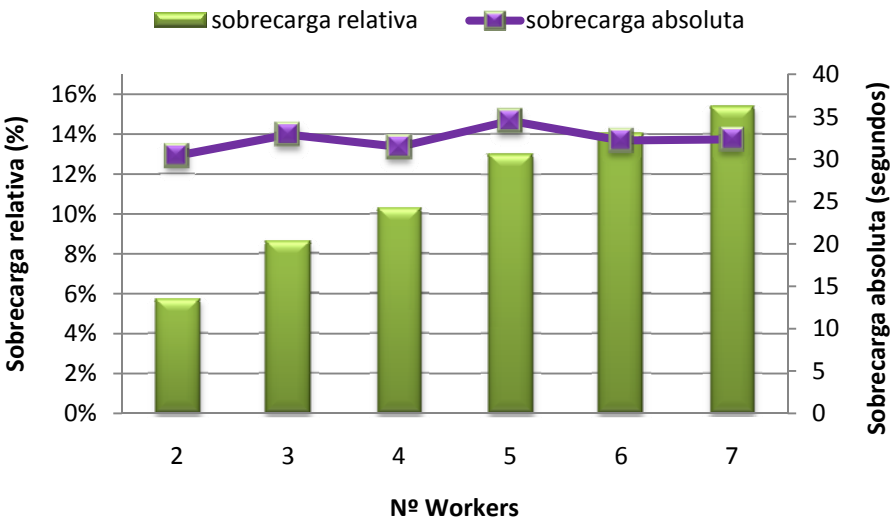

Figura 5.3 Sobrecarga relativa (%) y absoluta (segundos) introducida por MATE (sin aplicar sintonización) en la ejecución de Xfire.

El tercer escenario de ejecución aplica todo el funcionamiento de MATE sobre Xfire. Los resultados obtenidos en este entorno nos permite concluir que, una vez analizada y teniendo en cuenta la sobrecarga introducida por MATE, se observa una mejora de rendimiento cuando se aplica la estrategia de balanceo de carga a la aplicación Xfire. A lo largo de la ejecución de la aplicación MATE va adaptando el factor de distribución de datos a las condiciones actuales de

balanceo de la aplicación generando una mejora en el rendimiento. Dicha mejora es especialmente patente cuando el número de workers es mayor, ya que la división del conjunto de tareas totales en batches de menor tamaño permite un mayor solapamiento de los procesos de cómputo y comunicación.

Para entender mejor el comportamiento de la aplicación, se realizó un estudio en profundidad del algoritmo de procesamiento del master y de los workers.

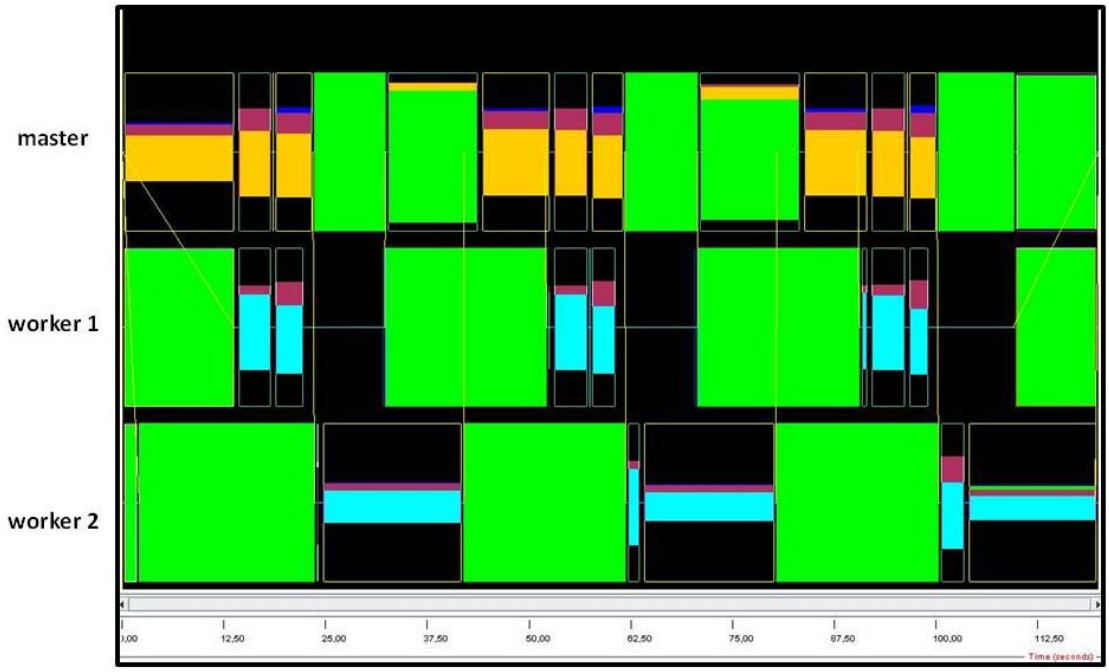

Figura 5.4 Comportamiento de la aplicación Xfire durante tres iteraciones.

El master a lo largo de la ejecución de la aplicación es el encargado de dividir la línea de fuego actual en fragmentos, preparar cada fragmento para su envío, enviar cada uno de ellos a un workers, recibir los resultados de los workers, combinarlos y finalmente generar la nueva línea de fuego. Una traza de la ejecución, mostrada en la figura 5.4, nos permitió ver que este comportamiento del proceso master genera tiempos de espera en los procesos workers, especialmente las fases de preparación de los fragmentos para el envío y la combinación de los fragmentos tras su recepción, lo cual provoca que la aplicación se encuentre un poco desbalanceada.

Para el caso del proceso worker, se observó que independientemente de cuál sea el punto de la línea de fuego el cómputo que se realiza es el mismo. Por lo tanto no existen puntos o tareas más pesadas que otras y todos requieren la misma potencia de procesamiento. Además partimos de la premisa de que las ejecuciones se están realizando en un entorno homogéneo y controlado, por lo que no existen workers más lentos que otros ni tareas ejecutándose de otros usuarios que dificulten el procesamiento del los workers.

Por último, es de destacar que la ganancia introducida por la técnica de sintonización, en algunos casos no solo es capaz de reducir el *overhead* introducido por MATE, sino que también se iguala o mejora el tiempo de ejecución obtenido respecto a la ejecución de la versión original de Xfire. Estos resultados se muestran en la figura 5.5.

Estas conclusiones muestran por tanto la efectividad del proceso de sintonización realizado a partir del tunlet desarrollado e integrado en MATE.

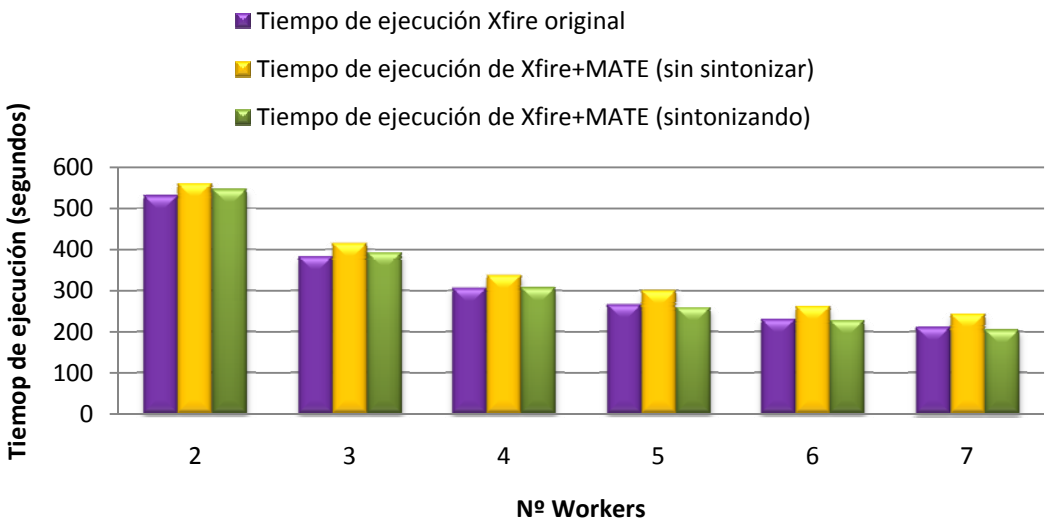

Figura 5.5 Comparativa de los tiempos de ejecución obtenidos para los distintos escenarios planteados.

Con el objetivo de poder observar de manera más clara, los beneficios que proporciona la estrategia de análisis desarrollado, y lograr unas mejoras de rendimiento más significativas, se decidió introducir un desbalanceo sintético en la aplicación, en concreto en los procesos worker. Este desbalanceo se ha simulado suponiendo que determinados puntos de la línea de fuego requieren un mayor cómputo que otros, es decir, constituyen tareas más pesadas.

Si se analiza la línea de fuego inicial, se puede observar que esta es una curva cerrada, en concreto una elipse. De tal modo, que para simular la existencia de tareas más pesadas, se ha establecido un umbral o frontera en dicha línea de fuego, mostrado en la figura 5.6, de manera que todos los puntos de la línea de fuego situados debajo de ese umbral corresponde a tareas que necesitan un mayor tiempo de cómputo.

Para reproducir esa idea el propósito era dejar ocioso cada proceso worker una cantidad de tiempo igual al tiempo de cómputo que el worker ha empleado para procesar su fragmento de la línea de fuego por la proporción de puntos de dicho fragmento que se sitúan por debajo del umbral. De esta manera, se logra introducir un desbalanceo variable a lo largo del tiempo, ya que depende de la línea de fuego y ésta se va reconstruyendo en cada iteración.

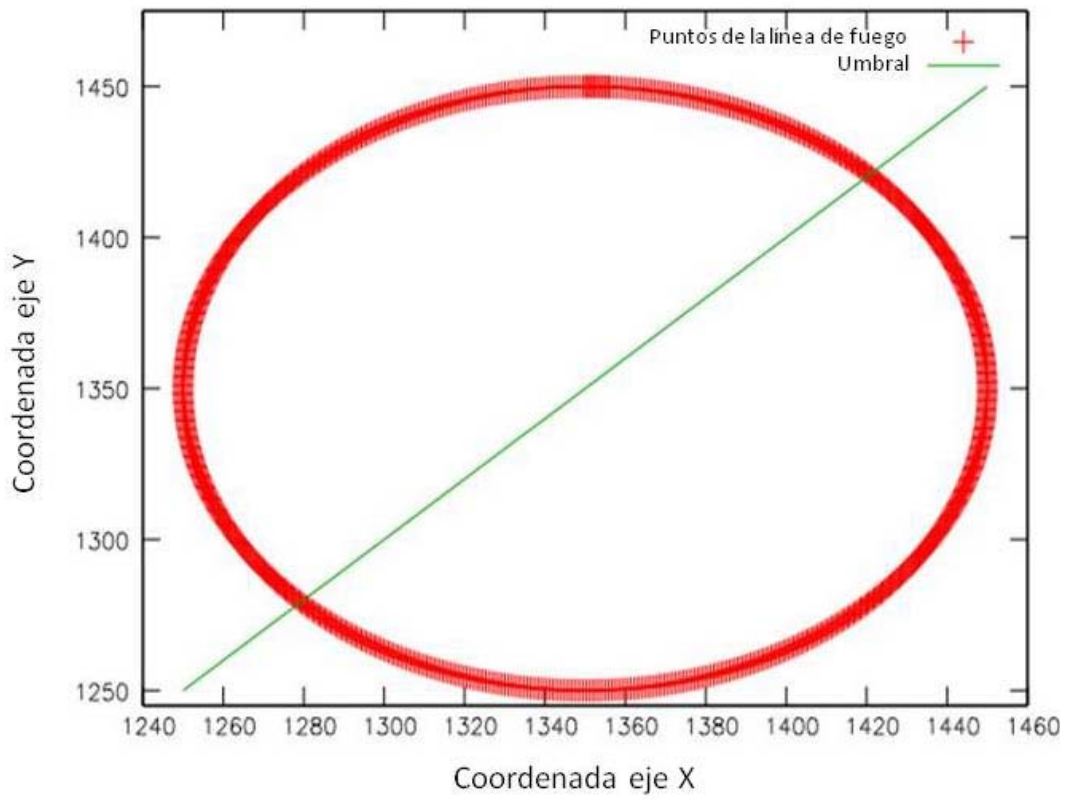

Figura 5.6 Estructura de la línea de fuego inicial y el umbral de desbalanceo empleado en las pruebas experimentales con Xfire.

Los resultados en cuanto a tiempo de ejecución obtenidos ejecutando Xfire en su versión original y Xfire siendo sintonizada a través del balanceo de carga para distintas combinaciones de número de workers introduciendo la desbalanceo sintético se muestran en la tabla 5.6.

Número de workers	**2**	**3**	**4**	**5**	**6**	**7**
Xfire+desbalanceo sintético	648,28	460,34	393,37	322,82	276,17	252,74
Xfire+MATE (sintonizando)+desbalanceo sintético	629,63	453,16	360,03	290,23	246,97	220,8
Ganancia %	2,9	1,5	8,4	10,09	10,57	12,63

Tabla 5.6 Tiempos de ejecución de Xfire considerando distinto número de workers con y sin desbalanceo sintético en los workers, y de Xfire bajo MATE con desbalanceo sintético (en segundos). Ganancia sobre el tiempo de ejecución de Xfire+desbalanceo sintético obtenida aplicando la sintonizción

Como se puede observar en la tabla, el tiempo de ejecución de Xfire aumenta cuando la carga es inyectada. En este caso, gracias al desbalanceo introducido, la mejora que proporciona la técnica de sintonización desarrollada en este trabajo es más acusada ya que dicho desbalanceo es en general corregido debido a que MATE detecta los cambios en las condiciones del sistema y adapta los factores de partición de los datos para distribuir el trabajo. Como se puede observar la ganancia aumenta conforme aumenta el número de workers, lo cual se debe a que el mayor

particionado de datos, permite que haya más solapamiento entre los procesos de cómputo y comunicación al haber un mayor número de workers. Por tanto, sería interesante realizar en futuros estudios estas pruebas aumentando el número de workers con el objetivo de observar si nuestra técnica de balanceo sigue siendo efectiva.

Capítulo 6

Conclusiones y trabajo futuro

6.1 Conclusiones

La computación paralela/distribuida proporciona la potencia necesaria para resolver problemas complejos. El rendimiento de las aplicaciones escritas para tales entornos de ejecución es un aspecto muy relevante.

Hoy en día existen distintas aproximaciones y herramientas que ayudan al usuario en el proceso de mejora de rendimiento proporcionándoles la información suficiente y apropiada sobre el comportamiento de la aplicación. Una de estas aproximaciones es la sintonización dinámica cuya principal característica es que realiza todo el proceso de análisis de prestaciones de aplicaciones paralelas/distribuidas de forma dinámica; es decir, monitoriza la aplicación para obtener información sobre su comportamiento, identifica los cuellos de botella y realiza las modificación de los parámetros críticos de la aplicación para mejorar el rendimiento, todo esto en tiempo de ejecución. Una de las herramientas que se sitúan bajo este enfoque es MATE, la cual ha sido el eje del presente trabajo.

El principal objetivo de esta investigación era sintonizar dinámicamente mediante MATE una aplicación MPI empleada en computación de altas prestaciones que siga un paradigma Master/Worker.

Para conseguir el fin propuesto, el trabajo comenzó estudiando otras aproximaciones y herramientas conocidas en el campo de la monitorización, análisis y sintonización de rendimiento de aplicaciones. Este análisis sirvió para observar como otros grupos o centros de

investigación afrontan el problema de mejora de rendimiento, y poder contextualizar nuestro trabajo claramente en el área de la sintonización dinámica.

Seguidamente, se procedió a estudiar la herramienta de sintonización dinámica MATE. Este estudio ha permitido tener una visión conceptual clara de la arquitectura que presenta la herramienta así como adquirir conocimientos sobre la funcionalidad de cada uno de los componentes de la citada arquitectura. Una tarea compleja fue lograr la comprensión global de cómo MATE realiza de forma dinámica todo el proceso de análisis de rendimiento, ya que para ello fue necesaria una inmersión en el código fuente de MATE. Este cometido llevó a tener el primer contacto con la técnica de instrumentación dinámica en la que se basa MATE para la realización del proceso de análisis en tiempo de ejecución.

El proceso de sintonización que se pretendía realizar en este trabajo estaba encaminado a solventar los cuellos de botella que presentan las aplicaciones Master/Worker. Para ello, MATE necesita poseer el conocimiento sobre dichos problemas de rendimiento. Los modelos de rendimiento constituyen la base del conocimiento empleado por MATE para conducir el proceso de análisis, determinando la información que se necesita recopilar durante la ejecución (*puntos de medida*), como evaluar la información recogida (*funciones de rendimiento*) y que cambios se necesitan para sintonizar la aplicación (*puntos/acciones/sincronizaciones de sintonización*). Para poder integrar este conocimiento en MATE, el siguiente paso fue el estudio de un modelo de rendimiento para aplicaciones Master/Worker, cuyo principal objetivo es resolver los problemas de rendimiento que en ellas se presentan aplicando una estrategia basada en dos fases: una primera fase en la se emplea una estrategia dinámica para el balanceo de carga y una segunda para adaptar el número de workers teniendo en cuenta las características actuales en las que se encuentra el sistema.

Por tanto, una vez que se estuvo el conocimiento sobre la herramienta de sintonización y estudiados los problemas de rendimiento que se desean resolver y su representación en el modelo de rendimiento, el último paso para lograr el objeto de esta investigación, fue el diseño y desarrollo del tunlet para ser integrado en MATE. El tunlet es el componente software o librería que constituyen el mecanismo inteligente empleado por MATE en la fase de análisis. Cada tunlet define e implementa una particular técnica de sintonización, de tal modo que en nuestro caso de estudio, el tunlet que se ha diseñado plantea la lógica de análisis necesaria para aplicar el modelo de rendimiento estudiado para aplicaciones Master/Worker. En este punto es de destacar que los tunlets desarrollados en anteriores trabajos de investigación implementaban una única técnica de sintonización, sin embargo, en nuestro caso de estudio, el tunlet creado presenta una complejidad más elevada ya que el conocimiento representado en él integra las dos estrategias de sintonización representadas en el modelo de rendimiento. La aplicación elegida

para ser sintonizada mediante la lógica integrada en el tunlet es un simulador de incendios forestales desarrollado bajo un paradigma Master/Worker, denominado Xfire.

Para el desarrollo del modelo de rendimiento en MATE, en el capítulo 5 se propone una metodología para la obtención del tunlet. Siguiendo esta metodología se ha logrado establecer:

- *Aquellos valores y variables de Xfire que se requerían para interpretar los parámetros del modelo de rendimiento*. Se determinó que las variables en Xfire que debían ser modificadas para mejorar el rendimiento eran: 1) el factor de partición adaptativo que indica el tamaño de los subconjuntos en los que será dividida la línea de fuego y 2) el número de workers de la aplicación. Además se dedujo cómo obtener todos los parámetros necesarios para evaluar la función analítica del modelo de rendimiento.

- *Los eventos que se debían capturar y la información asociada a los mismos.* Los eventos constituyen el mecanismo empleado por MATE para recopilar información sobre el comportamiento de la aplicación. Por tanto, nuestro trabajo fue determinar en que funciones de Xfire debían ser insertados los eventos para conseguir recopilar toda la información requerida.

- *Identificar los distintos procesos que participan en la ejecución de Xfire.* En este caso, al tratarse de una aplicación Master/Worker, únicamente había dos actores participando en la aplicación: procesos workers y un master.

El procedimiento que describe la metodología presentada estimamos que es el más adecuado para culminar con un tunlet que guie el proceso de sintonización deseado, de tal manera que puede ser generalizable para la sintonización bajo MATE de cualquier aplicación empleando cualquier modelo de rendimiento adecuado a ésta.

Hay que destacar que para poder aplicar sobre Xfire las técnicas de sintonización estudiadas fue necesario el estudio y la adaptación de dicha aplicación a las características requeridas por el modelo de rendimiento; en concreto las modificaciones se han realizado en la lógica de procesamiento del proceso master en la fase de distribución de datos entre workers. Si Xfire hubiese sido realizada empleando un framework conocido de diseño y programación paralela, el proceso de diseño del tunlet hubiera tenido una complejidad menos elevada. Además, dicho tunlet podría ser usado para mejorar el rendimiento de otra aplicación, que presente el mismo paradigma que Xfire y haya sido construida empleando el mismo framework.

Finalmente, una vez desarrollado el tunlet e integrado en MATE, procedimos a realizar pruebas experimentales. Hay que comentar que dichas pruebas, por motivos de tiempo, sólo cubrieron el estudio de rendimiento obtenido mediante la aplicación de la estrategia de balanceo

de carga sobre Xfire. Se plantearon varios escenarios de ejecución, con distintas combinaciones del número de workers, con el objetivo de obtener conclusiones sobre la sobrecarga que introduce MATE en la ejecución de la aplicación y sobre la ganancia obtenida en cuanto a rendimiento al aplicar las técnicas de sintonización.

Con respecto a la sobrecarga generada por MATE, se concluyó que ésta es constante independientemente del número de workers que participa en la ejecución de la aplicación, ya que la cantidad de instrumentación insertada y el número de eventos generados por proceso es siempre el mismo, y además estos son generados en paralelo.

Por otro lado, el análisis de la ganancia obtenida en el tiempo de ejecución de Xfire cuando ésta es ejecutada bajo el control de MATE muestra que las mejoras en el rendimiento aumentan conforme aumenta el número de workers, lo cual se debe a que el división adaptativa y dinámica del conjunto de tareas que realiza la técnica de factorización desarrollada permite que se produzca un mayor solapamiento entre los procesos de cómputo y comunicación.

Tras estos resultados, finalmente podemos concluir que la técnica de sintonización implementada e integrada en MATE es efectiva ya que la ejecución de Xfire bajo el control dinámico de MATE ha permitido observar la adaptación del comportamiento de dicha aplicación a las condiciones actuales del sistema donde se ejecuta, obteniendo así una mejora de su rendimiento.

6.2 Trabajo futuro

En cuanto al trabajo futuro, queda pendiente la depuración de la técnica de sintonización que permite adaptar el número de workers en la aplicación empleando un índice de rendimiento que directamente permite relacionar rendimiento, en cuanto a tiempo de ejecución, con eficiencia en el uso de recursos.

Lograr el funcionamiento de esta estrategia será una labor compleja ya que se necesita crear y eliminar procesos worker de forma dinámica en tiempo de ejecución; de modo que el estudio de la lógica de implementación que permita obtener esta funcionalidad se plantea como un gran reto.

Una vez estén funcionando de manera coordinada las dos técnicas de sintonización planteadas en el presente trabajo, se pretende realizar experimentación a partir de la cual poder obtener conclusiones sobre el comportamiento que presenta el tunlet desarrollado y las posibles influencias mutuas entre dichas estrategias de sintonización.

También se pretender llevar a cabo pruebas experimentales en entornos de cómputo que presenten un mayor número de nodos. Estas pruebas nos permitirán obtener de nuevo conclusiones sobre el comportamiento del tunlet, y además nos posibilitarán estudiar las características de escalabilidad de MATE. Tal y como se comentó en la sección 3.6 del capítulo 3, la escalabilidad de MATE se encuentran reducidas cuando aumenta el número de máquinas involucradas en la ejecución de la aplicación ya que el análisis centralizado que MATE lleva a cabo se convierte en un cuello de botella que hace que la detección de los problemas de rendimiento no sea rápida y por tanto los problemas de rendimiento no se resuelvan de manera adecuada.

Las conclusiones que se obtenga sobre el estudio de la escalabilidad, nos posibilitarán sentar las bases para comenzar con el trabajo futuro de la tesis doctoral que se centra en estudiar y mejorar la escalabilidad de MATE. Esta mejora inicialmente se centra en el empleo de un esquema de comunicación jerarquizado en la arquitectura de MATE y en un mecanismo de análisis de rendimiento distribuido. El fin que se persigue es poder hacer eficiente y útil el uso de MATE en el ámbito de la computación de altas prestaciones.

Bibliografía

[1] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*.: Pearson Addison Wesley, 2003.

[2] R. Jain, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*.: Wiley-Interscience, New York, 1991.

[3] W. E. Nagel, A. Arnold, M. Weber, H.-Ch. Hoppe, and K. Solchenbach, "VAMPIR: Visualization and Analysis of MPI Resources," *Supercomputer*, vol. 12, pp. 69-80, 1996.

[4] J. Jorba, T. Margalef, and E. Luque, "Search of Performance Inefficiencies in Message Passing Applications with KappaPI 2 Tool," in *Proceedings PARA Conference*, 2006, pp. 409-419.

[5] C. Tapus, I. Chung, and J.K. Hollingsworth, "Active Harmony: Towards Automated Performance Tuning," in *Proceedings from the Conference on High Performance Networking*, 2003, pp. 1-11.

[6] R.L. Ribler, H. Simitci, and D.A. Reed, "Autopilot Performance-Directed Adaptive Control System," in *Proceedings Future Generation Computer Systems Conference*, 1997, pp. 175-187.

[7] A. Morajko, "Dynamic Tuning of Parallel/Distributed Applications," Universidad Autónoma de Barcelona, Barcelona, Tesis doctoral 2003.

[8] J. K. Hollingsworth, "An API for Runtime Code Patching," *The International Journal of High Performance Computing Applications*, vol. 14, pp. 317-329, 2000.

[9] E. Cesar, A. Moreno, J. Sorribes, and E. Luque, "Modeling master/worker applications for automatic performance tuning," *Parallel Comput.*, vol. 32, no. 7, pp. 568-589, 2006.

[10] J Jorba, T Margalef, and E Luque, "Simulation of Forest Fire Propagation on Parallel and Distributed PVM Platforms," in *Proceedings of the 8th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 2001, pp. 386-392.

[11] P.H. Worley. (Accedida Junio 2010) Portable Instrumentation Library for MPI (MPICL). [Online]. http://www.netlib.org/mpicl/

[12] M. T. Heath and J. E. Finger. (Accedida Junio 2010) ParaGraph: A Performance Visualization Tool for MPI. [Online]. http://www.csar.illinois.edu/software/paragraph/

[13] P.H. Worley, "A new PICL trace file format," Oak Ridge National Laboratory, Oak Ridge, TN, ORNL Technical Report ORNL/TM-12125 September 1992.

[14] A. Geist, T.M. Heath, B.W. Peyton, and P.H. Worley, "A User's Guide to PICL: A Portable Instrumentation Communication Library," ORNL Technical Report ORNL/TM-11616, October 1990.

[15] D.A. Reed et al., "Scalable Performance Analysis: The Pablo Performance Analysis Environment," in *Proceedings of the Scalable parallel libraries conference*, 1993, pp. 104-113.

[16] R.A. Aydt, "SDDF: The Pablo Self-Describing Data Format," University of Illinois at Urbana Champaign, Department of Computer Science, Technical Report. 1993.

[17] D. Reed et al., "Scalable Performance Environments for Parallel Systems," in *Proceedings of the Sixth Distributed Memory Computing Conference*, 1991, pp. 562-569.

[18] H. Brunst, H.-C Hoppe, W. E. Nagel, and M. Winkler, "Performance Optimization for Large Scale Computing: The Scalable VAMPIR Approach," in *Proceedings International Conference on Computational Science*, 2001, pp. 751-760.

[19] S.S. Shende and A.D. Malony, "The Tau Parallel Performance System," *Int. J. High Perform. Comput. Appl.*, vol. 20, no. 2, pp. 287-311, 2006.

[20] F. Wolf and B. Mohr, "Automatic performance analysis of hybrid MPI/OpenMP applications," *Journal of Systems Architecture*, vol. 49, no. 10-11, pp. 421-439, 2003.

[21] M. S. Müller et al., "Developing Scalable Applications with Vampir, VampirServer and VampirTrace," *Parallel Computing: Architectures, Algorithms and Applications*, vol. 38, pp. 637-644, 2007.

[22] H. Brunst, D.K. Müller, M.S. Muller, and E.N. Wolfgang, "Tools for scalable parallel program analysis: Vampir NG, MARMOT, and DeWiz," *Int. J. Comput. Sci. Eng.*, vol. 4, no. 3, pp. 149-161, 2009.

[23] M. Geimer et al., "The SCALASCA Performance Toolset Architecture," in *Proceedings International Workshop on Scalable Tools for High-End Computing (STHEC)*, 2008, pp. 51-65.

[24] Brian J. N., "The Scalasca performance toolset architecture," *Concurrency and Computation: Practice and Experience*, vol. 22, pp. 702-719, 2010.

[25] K. Fürlinger and M. Gerndt, "Periscope: Performance Analysis on Large-Scale Systems," *Innovatives Supercomputing in Deutschland*, vol. 3, no. 2, pp. 26-29, 2005.

[26] M Gerndt, K. Furlinger, and E. Kereku, "Periscope: Advanced Techniques for Performance Analysis," in *Proceedings PARCO Conference*, 2005, pp. 15-26.

[27] S. Benedict, V Petkov, and M. Gerndt, "PERISCOPE: An Online-based Distributed Performance Analysis Tool," in *Proceedings 3rd International Workshop on Parallel Tools for High Performance*, 2009.

[28] B. Mohr, D. Brown, and A.D. Malony, "TAU: A Portable Parallel Program Analysis Environment for pC++," in *Proceedings CONPAR*, 1994, pp. 29-40.

[29] TAU (Tuning and Analysis Utilities). (Accedida en Junio 2010) [Online]. http://www.cs.uoregon.edu/research/tau/home.php

[30] V. Pillet, J. Labarta, T. Cortes, and S. Girona, "PARAVER: A Tool to Visualize and Analyze Parallel Code," Departament d'arquitectura de Computadors, Barcelona, Technical Report 1995.

[31] J. Labarta, S. Girona, and T. Cortes, "Analyzing Scheduling Policies Using Dimemas," Barcelona, Technical Report 1997.

[32] A. Espinosa, T. Margalef, and E. Luque, "Automatic detection of parallel program performance problems," in *SPDT '98: Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, New York, NY, USA, 1998, pp. 149.

[33] Barton P. Miller et al., "The Paradyn Parallel Performance Measurement Tools," *IEEE COMPUTER*, vol. 28, pp. 37-46, 1995.

[34] P.C. Roth, D.C. Arnold, and B.P. Miller, "MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools," in *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, Washington, DC, USA, 2003, pp. 21.

[35] R. Ribler, J. Vetter, H. Simitci, and D.A. Reed, "Autopilot: Adaptive Control of Distributed Applications," in *Proceedings of the 7th IEEE Symposium on High-Performance Distributed Computing*, 1998, pp. 172-179.

[36] A. Tiwari, V. Tabatabaee, and J.K. Hollingsworth, "Tuning parallel applications in parallel," *Parallel Computing*, vol. 35, no. 8-9, pp. 475-492, 2009.

[37] G. D. Riley and J. R. Gurd, "Towards Performance Control on the Grid," *Philosophical Transactions: Mathematical, Physical and Engineering Sciences*, vol. 363, no. 1833, pp. 1793-1805, 2005.

[38] M. Hussein, K. Mayes, M. Luján, and J. Gurd, "Adaptive performance control for distributed scientific coupled models," in *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*, New York, NY, USA, 2007, pp. 274-283.

[39] K. Chen, K. R. Mayes, and J. R. Gurd, "Autonomous performance control of distributed applications in a heterogeneous environment," in *Autonomics '07: Proceedings of the 1st international conference on Autonomic computing and communication systems*, ICST, Brussels, Belgium, Belgium, 2007, pp. 1-5.

[40] A. Morajko, P. Caymes-Scutari, T. Margalef, and E. Luque, "MATE: Monitoring, Analysis and Tuning Environment for parallel/distributed applications," *Concurr. Comput. : Pract. Exper.*, vol. 19, no. 11, pp. 1517-1531, 2005.

[41] A., Morajko, O., Jorba, J., Margalef, T., Luque, E. Morajko, "Dynamic Performance Tuning of Distributed Programming Libraries," in *ICCS'03: Proceedings of the 2003 international conference on Computational science*, vol. 2660, Melbourne, Australia, 2003, pp. 191-200.

[42] A. Morajko, O. Morajko, T. Margalef, and E. Luque, "MATE: Dynamic Performance Tuning Environment," in *Proceedings Euro-Par Conference*, 2004, pp. 98-106.

[43] G. Costa, J. Jorba, A. Morajko, T. Margalef, and E. Luque, "Performance models for dynamic tuning of parallel applications on Computational Grids," in *Proceedings 2008 IEEE International Conference on Cluster Computing*, Tsukuba, 2008, pp. 376-385.

[44] Dyninst Library. (Accedido en Mayo 2010) [Online]. http://www.dyninst.org

[45] P. Caymes-Scutari, "Extending the usability of a Dinamic tuning environment," Universidad Autónoma de Barcelona, Barcelona, Tesis Doctoral 2007.

[46] P. Caymes-Scutari, A. Morajko, T. Margalef, and E. Luque, "Scalable dynamic Monitoring, Analysis and Tuning Environment for parallel applications," *J. Parallel Distrib. Comput.*, vol. 70, no. 4, pp. 330-337, 2010.

[47] I. Banicescu and V. Velusamy, "Load Balancing Highly Irregular Computations with the Adaptive Factoring," in *Proceedings IPDPS Conference*, 2002.

[48] S. F. Hummel, E. Schonberg, and L. E. Flynn, "Factoring: A Method for Scheduling Parallel Loops," *Commun. ACM*, vol. 35, no. 8, pp. 90-101, 1992.

[49] R.C. Rothermel, "How to Predict the Sperad and Intensity of Forest and Range Fires," , vol. INT-143, 1983, pp. 1-5.

[50] Fire Software. (Accedido en mayo de 2010.) [Online]. http://fire.org

[51] J.C.S. André, E. Luque, and D.X. Viegas, "Application of parallel processing to the simulation of forest fire propagation," in *Proceedings International Conferencie on Forest Fire Research*, vol. II, Coimbra, Portugal, 1998.

[52] J.C.S. André and D.X. Viegas, "A Strategy to Model the Average Fireline Movement of a light-to-medium Intensity Surface Forest Fire," in *Proc. of the 2nd International Conference on Forest Fire Research*, 1994, pp. 221-242.

[53] J.C.S André, "A theory on the propagation of surface forest fire fronts," Universidade de Coimbra, Portugal, Tesis doctoral 1996.

[54] A. Morajko, P. Caymes-Scutari, T. Margalef, and E. Luque, "Automatic Tuning of Data Distribution Using Factoring in Master/Worker Applications," in *Proceedings International Conference on Computational Science*, 2005, pp. 132-139.