Análisis automático de prestaciones de aplicaciones paralelas basadas en paso de mensajes

Departament
d'Arquitectura de Computadors
i Sistemes Operatius

Tesis presentada por Josep Jorba Esteve para optar al grado de Doctor por la Universitat Autònoma de Barcelona

Barcelona, Febrero 2006

Análisis automático de prestaciones de aplicaciones paralelas basadas en paso de mensajes

Memoria presentada por Josep Jorba Esteve para optar al grado de Doctor en Informática por la Universitat Autònoma de Barcelona. Trabajo realizado en el Departament d'Arquitectura i Sistemes Operatius de l'Escola Tècnica Superior d'Enginyeria de la Universitat Autònoma de Barcelona, bajo la dirección del Dr. Tomàs Margalef Burrull.

Barcelona, febrero 2006

Director Tesis

Fdo. Tomàs Margalef Burull

Agradecimientos

Un buen grupo de personas han hecho este trabajo posible. Deseo expresar mi sincera gratitud a todas ellas por estar aquí en los momentos necesarios, por el trabajo compartido y por las ayudas prestadas.

Primero de todo dar las gracias a Tomàs Margalef por la dirección de este trabajo, por sus recomendaciones y su constante ayuda. A nuestro director Emilio Luque, por poner los medios necesarios para desarrollar este trabajo, y crear un ambiente adecuado, y estimulante, de trabajo en el grupo, sin el cual esta tesis no hubiese sido posible.

Gracias a Fernando, Eduardo y Juan Carlos por compartir esas interesantes conversaciones, con un estimulante café por medio, para intentar arreglar el mundo en los ratos libres.

Agradecer a los compañeros de ajedrez: Fernando, Xiao, Tomàs, y Miquel las interesantes partidas, y por dejarse ganar de vez en cuando, y así permitir acrecentar la autoestima propia como jugador. Siempre quedara cierto rencor por las veces que no se dejaron.

A Remo por las interminables discusiones teórico-técnicas de todo tipo, que me forzaban a mejorar cada día, a aprender algo más, y a descubrir que jamas acabaría sabiendo todo lo necesario.

También querría dar las gracias al resto de mis colegas del departamento de Arquitectura y sistemas operativos. Así como a todas aquellas personas que pasaron por el grupo, dejaron un grato recuerdo, y un tiempo compartido para no olvidar.

Pedir perdón a mi estimada Marta por el tiempo dedicado a este proyecto, y por otra parte no dedicado a ella; espero poder recuperarlo, y además agradecer su soporte moral durante todo este tiempo.

Prefacio

La constante evolución del cómputo de altas prestaciones ha permitido que su uso se extienda a un amplio rango de campos de aplicación tanto científicos, como comerciales. El constante avance en el hardware disponible (unido a la reducción en su coste), y la mejora de los paradigmas de programación ha permitido disponer de entornos muy flexibles para atacar algunos de los grandes problemas de cómputo antes inabordables.

Al campo clásico de la supercomputación, se le ha añadido, en las últimas décadas, la computación distribuida que ha puesto a disposición de un gran número de instituciones académicas y comerciales las capacidades de cómputo de altas prestaciones. Y en particular las soluciones de cómputo basadas en clusters de ordenadores personales, ampliamente disponibles por ser soluciones de bajo coste.

En la mayoría de las ocasiones estos avances de computación se han debido únicamente al avance en la tecnología del hardware, quedando relegado a un segundo término el software de sistema, ya sea el propio sistema operativo o los paradigmas de programación usados.

En la ultima década, la aparición de los paradigmas basados en paso de mensajes para la programación de altas prestaciones, con entornos como PVM y MPI, ha permitido incorporar esta programación a toda la gama de entornos hardware disponibles, desde las soluciones basadas en supercomputación hasta las más modestas basadas en clusters de variados tamaños (en número y capacidad de las máquinas que forman sus nodos).

Pero hay que tener en cuenta, que para obtener las capacidades de cómputo que esperamos de estos entornos, hay que asegurar que las aplicaciones han sido correctamente diseñadas y que sus prestaciones son satisfactorias. Esto implica que las tareas del desarrollador de la aplicación, no terminan cuando la aplicación está libre de fallos funcionales, sino que es necesaria la realización de un análisis de sus prestaciones, y la sintonización adecuada para alcanzar los índices esperados.

Ya sea como evolución lógica, o debido a la necesidad de abordar problemas con mayores requerimientos de cómputo, en los últimos tiempos se ha complementado la evolución en el hardware, con el análisis de prestaciones de los entornos software para intentar extraer de ellos los máximos picos de cómputo, o los mejores rendimientos de la relación cómputo/comunicaciones presentes en los paradigmas de paso por mensajes, o bien para aumentar la escalabilidad hardware/software de la solución empleada.

El proceso de sintonización de las aplicaciones requiere de un análisis de prestaciones, incluyendo la detección de los problemas de prestaciones, la identificación de sus causas y la modificación de la aplicación para mejorar su comportamiento.

La investigación en el ámbito de prestaciones de cómputo distribuido y paralelo, se ha centrado en la última década principalmente en el diseño de modelos básicos de problemas potenciales, así como de herramientas simples para su detección, y sobretodo, visualización en forma gráfica de los datos obtenidos de la ejecución de las aplicaciones.

Sin embargo, hay aún hoy una carencia relevante de herramientas prácticas de análisis de prestaciones, que permitan dar el retorno necesario al desarrollador de las aplicaciones. Los niveles de experiencia, y conocimiento, exigidos al desarrollador para poder examinar correctamente todos los datos referidos al comportamiento de la aplicación son muy significativos, y en muchas ocasiones incluso no asumibles en su totalidad.

En este contexto, se hace evidente, que un objetivo de las herramientas de análisis de prestaciones sería ayudar al desarrollador de aplicaciones, proporcionándole toda aquella información que pudiera ser de utilidad de forma directa, y que le permitiese mejorar su aplicación, con sugerencias aplicables a su metodología de trabajo, o a su código obtenido.

Los objetivos de esta tesis han sido ofrecer una arquitectura abierta para el desarrollo de herramientas de análisis automático de prestaciones, así como desarrollar una metodología de trabajo con estas herramientas que permita desde la integración del conocimiento de los expertos en el campo de las prestaciones, hasta la generación de sugerencias finales directas a los desarrolladores de las aplicaciones.

En el ámbito de nuestra arquitectura se han propuesto y analizado diversas técnicas para las diferentes fases de detección, clasificación, análisis de problemas, análisis de código fuente y generación de sugerencias al usuario final.

Por lo que se refiere a la clasificación y especificación del conocimiento referente a los problemas de rendimiento, hemos proporcionado diferentes visiones de las ineficiencias de prestaciones en las aplicaciones basadas en paso de mensajes, incorporando tanto los resultados de trabajos previos disponibles en la literatura, así como de la investigación previa del grupo. Esto nos ha permitido finalmente ofrecer una base de conocimiento relevante, aplicable a diferentes herramientas, y diversas aproximaciones al análisis de prestaciones.

El conocimiento se ha adoptado en forma de especificaciones abiertas (modificables y adaptables a otras herramientas análisis de prestaciones), proporcionadas mediante descripción estructural de patrones de problemas de prestaciones, así como de los diferentes casos de uso que presentaban estos problemas y/o ineficiencias, según las condiciones a evaluar, para determinar las sugerencias finales al usuario, dependiendo de las observaciones de cada problema concreto dentro de las aplicaciones analizadas.

Para demostrar nuestra metodología y la arquitectura propuesta, se ha imple-

mentado, como prueba de concepto, la herramienta KappaPI 2 sobre entornos de paso por mensajes, mediante las interfaces de programación PVM y MPI. Se ha comprobado la metodología en la práctica, aplicando la herramienta satisfactoriamente en diversos experimentos sobre benchmarks, kernels de algoritmos y aplicaciones científicas reales.
Esta tesis se ha organizado en los siguientes capítulos:

Capítulo 1 En el capítulo 1 se presenta una introducción a los sistemas paralelos y/o distribuidos. Se analizan diferentes modelos de arquitectura, en concreto en los clusters donde hemos centrado nuestros estudios. Se han analizado también los diferentes paradigmas de programación que se utilizan y se ha centrado el estudio sobre los de paso de mensajes, y en concreto en los entornos PVM y MPI objeto de este trabajo. Se introducen también algunos conceptos básicos (y clásicos) del análisis de prestaciones en sistemas paralelos.

Capítulo 2 Se analizan los conceptos relacionados con las prestaciones que usaremos a lo largo del trabajo. Se hace especial énfasis en las diferentes metodologías clásicas de monitorización, el tracing y el profiling, así como las diferentes consideraciones referentes a modelos y tiempos asociados a las comunicaciones presentes en los modelos de paso de mensajes analizados. También se realiza una recopilación de algunos ejemplos destacables de herramientas, de las que podríamos llamar de primera generación, por la recopilación de la información normalmente destinada a ofrecer una visualización gráfica del comportamiento de la aplicación. Se introduce el concepto de análisis automático, que se desarrollará en el siguiente capítulo, en contraposición a la visualización.

Capítulo 3 Examinamos el concepto de análisis automático de prestaciones, y las diferentes aproximaciones existentes en la literatura, tanto de tipo estático, como dinámico. Se analiza el uso de las diferentes aproximaciones utilizadas en las herramientas de análisis automático representativas disponibles de cada aproximación.

Capítulo 4 En este capítulo se describe la arquitectura que se ha concebido para el diseño de una herramienta de análisis automático de prestaciones, mostrando la funcionalidad de los diferentes componentes de la misma, y su implementación en nuestra herramienta de análisis automático de prestaciones: KappaPI 2. Siguiendo un esquema lineal, se analizan las diferentes fases de la arquitectura, teniendo en cuenta las ideas presentes y su implementación. Se describe el catálogo con los diferentes problemas de prestaciones que se pueden presentar en los entornos de paso de mensajes, así como la información necesaria para poder clasificar las ineficiencias que se encuentren en la ejecución de la aplicación considerando los casos y condiciones que puede presentar cada tipo de problema. Finalmente se analizan diferentes técnicas

empleadas en la herramienta para el análisis del código fuente que nos permiten obtener la información final para generar sugerencias sobre la mejora de la aplicación al usuario.

Capítulo 5 En este capítulo se presentan los resultados de la experimentación realizada, considerando algunos casos de estudio a los que se ha aplicado la herramienta. Esta experimentación ha permitido validar las diferentes fases del análisis de prestaciones, y comprobar la bondad de los resultados obtenidos sobre un conjunto variado de experimentos, incluyendo aplicaciones sintéticas, benchmarks paralelos, y aplicaciones científicas paralelas reales.

Capítulo 6 Finalmente, el capítulo 6 resume las conclusiones de esta tesis con los principales resultados, las experiencias derivadas del estudio realizado, la aplicabilidad de los métodos propuestos, y las lineas de continuación que se han empezado a desarrollar o pretendemos abordar en un futuro.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

Este capítulo muestra una perspectiva de los sistemas paralelos y/o distribuidos. Se presentan los diferentes modelos arquitecturales, examinando en concreto los clusters donde hemos centrado nuestros estudios. Se analizan también los diferentes paradigmas de programación que se utilizan, y se centra la atención sobre los entornos de paso de mensajes, en concreto en PVM y MPI, objeto del estudio de análisis de prestaciones de este trabajo. También se introducen también algunos conceptos básicos (y clásicos) del análisis de prestaciones en sistemas paralelos.

1.1. Introducción a los sistemas paralelos distribuidos

Durante las ultimas décadas, la computación de altas prestaciones se ha utilizado para resolver los más complejos problemas de diversos campos de la ciencia y el mundo económico. Cuanto más han evolucionado el hardware y el software orientado a los sistemas paralelos, mayores han sido los problemas que se han conseguido abordar, y se han abierto las puertas a problemas hasta ahora intratables.

La computación paralela y/o distribuida comporta también unas implicaciones económicas y sociales importantes. Sus avances se reflejan a menudo en la competición industrial para producir mejores productos, y en algunos campos científicos como biología, química y medicina por el bien social común en la mejor comprensión de mecanismos de enfermedades, de drogas y medicamentos. También, se ha iniciado el camino para grandes hitos futuros como la exploración del conocimiento de los mecanismos del ADN humano, el análisis de la estructura de proteínas, y otros campos como la predicción de desastres naturales como terremotos, inundaciones y tsunamis. Las situación actual y las posibles mejoras en el cómputo de altas prestaciones nos abre toda una nueva serie de caminos para la ciencia y la economía.

Pero para que todo esto sea posible, hay que continuar explorando las mejoras posibles a realizar en los sistemas paralelos y/o distribuidos. El hardware de altas prestaciones ha tenido una escalabilidad inmensa desde sus inicios, siguiendo la

ley de Moore, o incluso superándola durante mucho tiempo, debido a la irrupción de los sistemas basados en cluster y Grid. Pero a su vez, el software se ha visto frenado por diversas razones, tales como la complejidad de los sistemas hardware involucrados, la complejidad de su desarrollo, de su implementación, de su testeo y de su mantenimiento.

Desgraciadamente, no es fácil en aplicaciones paralelas reales sostener unos buenos índices de prestaciones que alcancen los porcentajes teóricos que podemos obtener. Las razones de esta limitación de las aplicaciones (y de sus desarrolladores) está en las complejas iteraciones que las aplicaciones soportan con el sistema físico, el software de sistema (operativo y librerías), las interfaces de programación usadas (cada vez de más alto nivel) y los algoritmos implementados.

Comprender todas estas interacciones, y como mejorarlas, es crucial para optimizar las prestaciones que podamos obtener de las aplicaciones y alcanzar la mejor utilización y productividad de los recursos hardware disponibles, para maximizar las prestaciones de los sistemas disponibles.

1.2. Modelos arquitecturales

Los computadores paralelos (y los sistemas distribuidos) disponen de múltiples procesadores que son capaces de trabajar conjuntamente en una o varias tareas a la vez para resolver un problema computacional.

Hay muchas maneras diferentes en que los sistemas distribuidos y paralelos pueden ser construidos y clasificados en diferentes taxonomías. En esta sección presentamos varias clasificaciones basadas en diferentes conceptos.

En función de los conjuntos de instrucciones y datos, y frente a la arquitectura secuencial que denominaríamos SISD (*Single Instruction Single Data*), podemos clasificar las diferentes arquitecturas paralelas en diferentes grupos (a esta clasificación se la suele denominar taxonomía de Flynn) [Akl89, Alm94, McB94, Ste96, Ste02]: SIMD (*Single instruction Multiple Data*), MISD (*Multiple Instruction Single Data*) y MIMD (*Multiple Instruction Multiple Data*), con algunas variaciones como la SPMD (*Single Program Multiple Data*)

- SIMD: Un solo flujo de instrucciones es concurrentemente aplicado a múltiples conjuntos de datos. En una arquitectura SIMD unos procesos homogéneos (con el mismo código) síncronamente ejecutan la misma instrucción sobre sus datos, o bien la misma operación se aplica sobre unos vectores de tamaño fijo o variable. El modelo es válido para procesamientos matriciales y vectoriales, siendo las máquinas con procesadores vectoriales un ejemplo de esta categoría.

- MISD: El mismo conjunto de datos es tratado de forma diferente por los procesadores. Son útiles en casos donde sobre el mismo conjunto de datos se deban realizar muchas operaciones diferentes. En la práctica no se

han construido máquinas de este tipo, por las dificultades en su concepción [Ste96, Ste02].

- MIMD: Paralelismo funcional y/o de datos. No solo distribuimos datos sino también las tareas a realizar entre los diferentes procesadores. Varios flujos (posiblemente diferentes) de ejecución son aplicados a diferentes conjuntos de datos. Esta categoría no es muy concreta, ya que existe una gran variedad de arquitecturas posibles con estas características, incluyendo máquinas con varios procesadores vectoriales, o sistemas de centenares de procesadores.

- SPMD: En paralelismo de datos [Hil86], utilizamos mismo código con distribución de datos. Hacemos varias instancias de las mismas tareas, cada uno ejecutando el código de forma independiente. SPMD puede verse como una extensión de SIMD o bien una restricción del MIMD. A veces suele tratarse más como un paradigma de programación (como veremos en la sección 1.4), en el cual el mismo código es ejecutado por todos los procesadores del sistema, pero en la ejecución se pueden seguir diferentes caminos en los diferentes procesadores.

Esta clasificación (taxonomía de Flynn) de modelos arquitecturales se suele ampliar para incluir diversas categorías de computadoras que no se ajustan totalmente a cada uno de estos modelos. Una clasificación extendida de Flynn podría ser la siguiente [Dun90]:

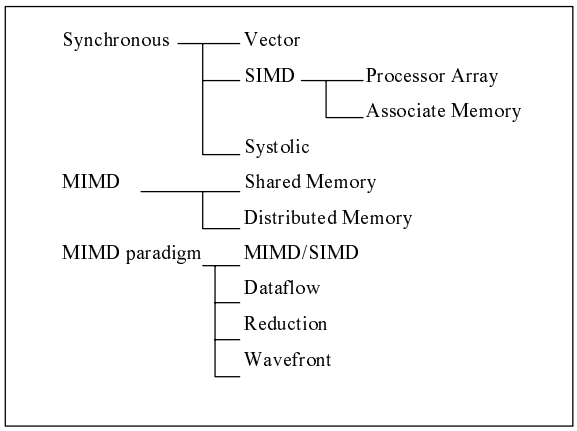

Figura 1.1: Clasificación de las arquitecturas paralelas

En esta clasificación (figura 1.1), se ha extendido la clásica para incluir desarrollos de máquinas con diseños arquitectónicos concretos. En particular, comentar la inclusión de las computadoras vectoriales en la categoría SIMD. Estas máquinas incorporan una CPU clásica (bajo el modelo von Neumann), pero con vectores como tipo de datos primitivo. Tenemos instrucciones del repertorio que permite

operar sobre todas las componentes del vector. Estas máquinas con uno o más procesadores vectoriales (en algunos casos MIMD), son las que clásicamente se han conocido con la denominación de supercomputadores (en particular las computadoras vectoriales del fabricante Cray).

Otra extensión de esta clasificación de los modelos arquitecturales es la basada en los mecanismos de control y la organización del espacio de direcciones [Kum94, Fos95, Cul99], en el caso de las arquitecturas MIMD, las de más amplia repercusión hoy en día:

- Sistemas MIMD con memoria compartida (Shared Memory-MIMD), o también denominados *Multiprocesadores*: En este modelo los procesadores comparten el acceso a una memoria común. Existe un único espacio de direcciones de memoria para todo el sistema, accesible para todos los procesadores. Los procesadores se comunican a través de esta memoria compartida. Los desarrolladores no tienen que preocuparse sobre la posición actual de almacenamiento de los datos, y todos los procesadores tienen el mismo espacio de acceso a los datos. Ejemplos de estos sistemas incluyen: SGU Challenge, estaciones de trabajo multiprocesador, Cray J90, Cray T90, Digital Alpha y Tera-MTA. En particular, cabe destacar la denominación comercial de sistemas SMP (*Symetric Multiprocessing*), refiriéndose a la combinación de MIMD y memoria (físicamente) compartida. Sus principales limitaciones son el número de procesadores dependiendo de la red de interconexión, siendo habitual sistemas entre 2-4 a 16 procesadores. Aunque también existe la opción, por parte de varios fabricantes, de unir varios de estos sistemas (formando un sistema híbrido), en lo que se conoce como SPP (*Scalable Parallel Processing*) o CLUMP (CLUsters of MultiProcessors) [Cap99, Cap00], para formar un sistema mayor. Como resumen, podemos decir que estas máquinas (SMP y SPP) ofrecen flexibilidad y fácil programación (por el esquema de memoria compartida), a costa de complejidad adicional en el hardware.

- Sistemas MIMD con memoria distribuida (Distributed Memory-MIMD), o también llamados *Multicomputadores*: En este modelo cada procesador ejecuta un conjunto separado de instrucciones sobre sus propios datos locales. La memoria, no centralizada, está distribuida entre los procesadores del sistema, cada uno con su propia memoria local donde posee su propio programa y los datos asociados. El espacio de direcciones de memoria no está compartido entre los procesadores. Una red de interconexión conecta los procesadores (y sus memorias locales), mediante enlaces (*links*) de comunicación, usados para el intercambio de mensajes entre los procesadores. Los procesadores intercambian datos entre sus memorias cuando se pide el valor de variable remotas. Ejemplos clásicos de estos sistemas incluyen: IBM SP, Thinking Machines CM5, Cray T3D, SGI Origin2000 y el ASCI Red. También podemos observar unas subdivisiones de este modelo MIMD de memoria distribuida, Por un lado: a) Una extensión natural de este modelo seria el uso de

una red de computadoras (evidentemente con una latencia más grande que la que presenta una red de interconexión dedicada). En este caso, cada nodo de la red es en sí mismo una computadora completa, pudiendo incluso operar de forma autónoma del resto de nodos. Dichos nodos pueden, además, estar distribuidos geográficamente en localizaciones distintas. A estos sistemas se les suele denominar clusters, y los analizaremos con más detalle posteriormente. b)Comercialmente, es habitual a los sistemas MIMD con memoria físicamente distribuida y red de interconexión dedicada especialmente diseñada, denominarlos sistemas MPP (*Massive Parallel Processing*). En éstos, el número de procesadores puede variar desde unos pocos a miles de ellos. Normalmente los procesadores (de un sistema MPP) están organizados formando una cierta topología (tanto física como lógica, como: anillo, árbol, malla, hipercubo, etc.), disponiendo de un red de interconexión entre ellos de baja latencia y gran ancho de banda.

En esta ultima clasificación es necesario tener en cuenta la diferenciación de la organización de la memoria, entre la organización física y la lógica. Podemos, por ejemplo, disponer de una organización física de memoria distribuida, pero lógicamente compartida. En el caso de los multiprocesadores podríamos hacer una subdivisión más entre sistemas fuertemente o débilmente acoplados.

En un sistema fuertemente acoplado, el sistema ofrece un mismo tiempo de acceso a memoria para cada procesador. Este sistema puede ser implementado a través de un gran módulo único de memoria, o por un conjunto de módulos de memoria de forma que puedan ser accedidos en paralelo por los diferentes procesadores. El tiempo de acceso a memoria (a través de la red de interconexión común) se mantiene uniforme para todos los procesadores. Tenemos un acceso uniforme a memoria (UMA).

En un sistema ligeramente acoplado, el sistema de memoria está repartido entre los procesadores, disponiendo cada uno de su memoria local. Cada procesador puede acceder a su memoria local y a la de los otros procesadores, pero el tiempo de acceso a las memorias no locales es superior a la de la local, y no necesariamente igual en todas. Tenemos un acceso no uniforme a la memoria (NUMA).

Por ultimo analizamos en está sección, como caso que merece especial atención, los clusters, que pueden verse como un tipo especial de sistema MIMD de memoria distribuida (DM-MIMD), y que será el objeto de nuestro estudio.

1.2.1. Clusters

Los clusters [Alm94, Die98] consisten en una colección de computadores (no necesariamente homogéneos) conectados por red(Gigabit) Ethernet, Fiber Channel, ATM, u otras tecnologías de red para trabajar concurrentemente en tareas del mismo programa. Un cluster está controlado por una entidad administrativa simple (normalmente centralizada) que tiene el control completo sobre cada sistema final. Como modelo de arquitectura son sistemas MIMD de memoria distribuida, aunque

la interconexión puede no ser dedicada, y utilizar mecanismos de interconexión simples de red local, o incluso a veces no dedicados exclusivamente al conjunto de máquinas. Básicamente es un modelo DM-MIMD, pero las comunicaciones entre procesadores, son habitualmente, varios ordenes de magnitud más lentas.

Esta combinación (homogénea o heterogénea) de máquinas, podemos utilizarla como computador paralelo, gracias a la existencia de paquetes software que permiten ver el conjunto de nodos disponibles como una maquina paralela virtual, ofreciendo una opción práctica, económica y popular hoy en día para aproximarse al cómputo paralelo. Algunas de las ventajas que podemos encontrar en estos sistemas son las siguientes:

- Cada máquina del cluster es en sí misma un sistema completo, utilizable para un amplio rango de aplicaciones de usuario. Pero estas aplicaciones, normalmente no consumen, de forma constante, un tiempo apreciable de CPU. Una posibilidad es usar, las máquinas del cluster, aprovechando estos tiempos (o porcentajes de tiempo) muertos para utilizarlos en las aplicaciones paralelas, sin que esto perjudique a los usuarios convencionales de las máquinas. Esto también nos permite diferenciar entre los clusters dedicados (o no), dependiendo de la existencia de usuarios (y por tanto de sus aplicaciones) conjuntamente con la ejecución de aplicaciones paralelas. Así mismo, también podemos diferenciar entre dedicación del cluster a una sola o varias aplicaciones paralelas simultaneas en ejecución.

- El aumento significativo de la existencia de los sistemas en red con un amplio mercado y su comercialización, ha hecho que el hardware para estos sistemas sea habitual y económico, de manera que podemos montar una máquina paralela virtual a bajo coste. A diferencia de las máquinas SMP y otros supercomputadores, es más fácil conseguir buenas relaciones de coste y prestaciones. De hecho, buena parte de los sistemas actuales [Top05] de altas prestaciones son sistemas en cluster, con redes de interconexión más o menos dedicadas.

- Los podríamos situar como una alternativa intermedia entre los sistemas comerciales basados en SMP y MPP. Sus prestaciones van mejorando en la medida que se desarrolla software especifico para aprovechar las características de los sistemas en cluster, así como que se introducen mejoras en las tecnologías de red local y de medio alcance.

- La computación en cluster es bastante escalable (en hardware). Es fácil construir sistemas con centenares o miles de máquinas. Por contra las SMP suelen estar limitadas en número de procesadores. De hecho, en los clusters, en la mayor parte de las ocasiones la capacidad se ve limitada por las tecnologías de red, debiendo ser las redes de interconexión las que puedan soportar el número de máquinas que queramos conectar, ya sea utilizando tecnologías de red generales (Ethernet, Fast o gigaethernet, ATM, etc...) o bien

redes de altas prestaciones dedicadas (Myrinet, SCI, etc). Además, los entornos software para cluster ofrecen una gran escalabilidad desde cómputo en pequeños clusters, como los sistemas de paso de mensajes, hasta sistemas de cómputo conectados a través de Internet, como son los sistemas Grid [Fos99].

- Aunque no es habitual disponer de hardware tolerante a fallos como es el caso de SMP y supercomputadores, normalmente se incorporan mecanismos software que invalidan (y/o substituyen por otro) el recurso que ha fallado.

Pero también hay una serie de problemas relevantes a considerar:

- Con contadas excepciones, el hardware de red no está diseñado para el procesamiento paralelo. La latencia típica es muy alta, y el ancho de banda es relativamente bajo comparado con sistemas SMP o supercomputadores.
- El software suele haber sido diseñado para máquinas personales. Tal es el caso del sistema operativo, y normalmente no ofrece posibilidades de control (y gestión) de clusters. En estos casos es necesario incorporar toda una serie de capas de servicios middleware sobre el sistema operativo, para hacer que los sistemas en cluster sean eficaces. En sistemas a mayor escala, es necesario incluir toda una serie de mecanismos de control y gestión adicionales, para el scheduling ([Boe03]) y monitorización de los sistemas, como es el caso en los sistemas Grid. En general esto supone una complejidad muy grande del software de sistema, que aumenta sensiblemente la complejidad total, y tiene una repercusión significativa sobre las prestaciones en estos sistemas.

1.3. Modelos de programación

Las aplicaciones paralelas consisten en una o más tareas que pueden comunicarse y cooperar para resolver un problema.

En cuanto a la creación de las aplicaciones paralelas, no hay una metodología claramente establecida, ni fija, debido a la fuerte dependencia de las arquitecturas de las máquinas que se usen, y los paradigmas de programación usados en su implementación.

Una metodología simple de creación de aplicaciones paralelas [Fos95], podría ser la que estructura el proceso de diseño en cuatro etapas diferentes: Partición, Comunicación, Aglomeración y Mapping (a veces a esta metodología se la denomina con el acrónimo PCAM). Las dos primeras etapas se enfocan en la concurrencia y la escalabilidad, y se pretende desarrollar algoritmos que primen estas características. En las dos últimas etapas, la atención se desplaza a la localidad y las prestaciones ofrecidas.

1. Partición: El cómputo a realizar y los datos a operar son descompuestos en pequeñas tareas. El objetivo se centra en detectar oportunidades de ejecución

paralela. Para diseñar una partición, observamos los datos del problema, determinamos particiones de estos datos, y finalmente se asocia el cómputo con los datos. A esta técnica se la denomina descomposición del dominio. Una aproximación alternativa consiste en la descomposición funcional, asignando diferentes cómputos o fases funcionales a las diferentes tareas. Las dos son técnicas complementarias que pueden ser aplicadas a diversos componentes o fases del problema, o bien aplicadas al mismo problema para obtener algoritmos paralelos alternativos.

2. Comunicación: Se determinan las comunicaciones necesarias (en forma de estructuras de datos necesarias, protocolos, y algoritmos), para coordinar la ejecución de las tareas.

3. Aglomeración: Las tareas y estructuras de comunicación de las dos primeras fases son analizadas respecto de las prestaciones deseadas, y los costes de implementación. Si es necesario, las tareas son combinadas en tareas mayores, si con esto se consigue reducir los costes de comunicación y aumentar las prestaciones.

4. Mapping: Cada tarea es asignada a un procesador de manera que se intentan satisfacer los objetivos de maximizar la utilización del procesador, y minimizar los costes de comunicación. El mapping puede especificarse de forma estática, o determinarlo en ejecución mediante métodos de balanceo de carga.

El resultado de esta metodología, dependiendo de los paradigmas y las arquitecturas físicas, puede ser una aplicación paralela, con un mapping estático entre tareas y procesadores a la hora de iniciar la aplicación, o bien una aplicación que crea tareas (y las destruye) de forma dinámica, mediante técnicas de balanceo de carga. Alternativamente, también podemos utilizar estrategias de tipo SPMD (ver sección 1.4.2) que crea exactamente una tarea por procesador, asumiendo que la etapa de aglomeración ya combina las tareas de mapping.

Una de las decisiones importantes en las aplicaciones paralelas, como resultado de las tareas anteriores, es precisamente el paradigma de programación que usaremos. Con cada uno de los paradigmas de programación, nos estamos refiriendo a una clase de algoritmos que tienen la misma estructura de control [Han93, Mol93], y que pueden ser implementados usando un modelo genérico de programación paralela.

En las próximas secciones introduciremos los modelos más importantes usados de programación paralela y de paradigmas de programación paralela (sección 1.4). Para los modelos de programación remarcamos como solucionan la distribución de código y la interconexión entre las unidades de ejecución y las tareas. Los modelos de programación paralela [Kum94, Fos95] están divididos entre los que veremos a continuación. Cualquiera de estos modelos de programación puede ser usado para implementar los paradigmas de programación paralela. Pero las prestaciones de

cada combinación resultante (paradigma en un determinado modelo) dependerán del modelo de ejecución subyacente (la combinación de hardware paralelo, red de interconexión y software de sistema disponibles).

1.3.1. Memoria compartida

En este modelo [Ben90, Cha94] los programadores ven sus programas como una colección de procesos accediendo a variables locales y un conjunto de variables compartidas. Cada proceso accede a los datos compartidos mediante una lectura o escritura asíncrona. Por tanto, como más de un proceso puede realizar las operaciones de acceso a los mismos datos compartidos en el mismo tiempo, es necesario implementar mecanismos para resolver los problemas de exclusiones mutuas que se puedan plantear, mediante mecanismos de semáforos o bloqueos.

En este modelo el programador ve la aplicación como una colección de tareas, que normalmente son asignadas a threads de ejecución en forma asíncrona. Los threads poseen acceso al espacio compartido de memoria, con los mecanismos de control citados anteriormente. En cuanto a los mecanismos de programación utilizados pueden usarse las implementaciones de threads en diferentes operativos, y los segmentos de memoria compartida, así como paralelismo implícito en algunos casos. En este último, se desarrollan aplicaciones secuenciales, donde se insertan directivas que permiten al compilador realizar paralelizaciones de código en diversas secciones.

OpenMP [Wol03] es una de las implementaciones más utilizadas para programar bajo este modelo, en sistemas de tipo SMP (o SH-MIMD). OpenMP (*Open Specifications for Multi Processing*) define directivas, y primitivas de librería para controlar la paralelización de bucles y otras secciones de un código en lenguajes como Fortran, C y C++. La ejecución se basa en la creación de thread principal, juntamente con la creación de threads esclavos cuando se entra en una sección paralela. Al liberar la sección se reasume la ejecución secuencial. OpenMP normalmente se implementa a partir librerías de bajo nivel de threads.

En OpenMP se usa un modelo de ejecución paralela denominado fork-join, básicamente el thread principal, que comienza como único proceso, realiza en un momento determinado una operación fork para crear una región paralela (directivas PARALLEL, END PARALLEL) de un conjunto de threads, que acaba mediante una sincronización por una operación join, reasumiéndose el thread principal de forma secuencial, hasta la siguiente región paralela. En este sentido, vemos un modelo explicito (no automático) del paralelismo, que ofrece al programador un control total sobre el proceso de paralelización. Todo el paralelismo de OpenMP se hace explicito mediante el uso de directivas de compilación que están integradas en el código fuente (Fortran, o C/C++). Las regiones paralelas suelen integrar diferentes construcciones de paralelismo (definidas por directivas), como la integración de compartición de iteraciones de bucles (DO, END DO) en los diferentes threads, secciones de trabajo (SECTION, END SECTION) que son repartidas entre los threads, o una porción secuencial (SINGLE, END SINGLE) que es realizada por

un único thread con objetivo de serializar el código. También se soportan (dependiendo de la implementación) la posibilidad de variar dinámicamente el número de threads en la región paralela, y el paralelismo anidado (anidar regiones paralelas), y sincronización explicita por si fuera necesaria un grado más fino de sincronización (la mayoría de las construcciones ya suponen una sincronización implícita al final de las mismas).

En caso de sistemas físicos paralelos de tipo híbrido, que soporten por ejemplo nodos SMP en cluster o de tipo SPP, suele combinarse la programación OpenMP con otros modelos como el de paso de mensajes.

1.3.2. Paralelismo de datos

El paralelismo de datos, es un paradigma en el cual [Alm94] operaciones semejantes (o iguales) son realizadas sobre varios elementos de datos simultáneamente, por medio de la ejecución simultanea en múltiples procesadores.

Este modelo [Hat91] es aconsejable para las aplicaciones que realizan la misma operación sobre diferentes elementos de datos. La idea principal es explotar la concurrencia que deriva de que la aplicación realice las mismas operaciones sobre múltiples elementos de las estructuras de datos.

Un programa paralelo de datos consiste en una lista de las mismas operaciones a realizar en una estructura de datos. Entonces, cada operación en cada elemento de datos puede realizarse en forma de una tarea independiente.

El paralelismo de datos es considerado como un paradigma de más alto nivel, en el cual al programador no se le requiere que haga explícitas las estructuras de comunicación entre los elementos participantes. Éstas son normalmente derivadas por un compilador que realiza la descomposición del dominio de datos, a partir de indicaciones del programador sobre la partición de los datos. En este sentido, hemos de pensar que el programa posee una semántica de programa secuencial, y solamente debemos pensar en el entorno paralelo para la selección de la distribución de los datos teniendo en cuenta la ayuda del compilador y las posibles directivas que le podamos dar para guiar el proceso.

El modelo de diseño de aplicaciones paralelas PCAM (mencionado previamente) sigue siendo aplicable, aunque en los lenguajes de paralelismo de datos, se realiza la primera fase directamente, por medio de construcciones implícitas y otras explícitas proporcionadas por el usuario, de cara a obtener una granularidad fina de computación [Alm94]. Un punto importante en esta fase, es identificar (mediante el compilador y la ayuda proporcionada por el usuario) particiones con un grado suficiente de concurrencia. En el caso del paralelismo de datos, las restantes fases del modelo de diseño PCAM, pueden realizarse de forma práctica gracias a directivas de compilador de alto nivel, en lugar de pensar en términos de comunicaciones explicitas y operaciones de mapping a nodos de cómputo.

La conversión del programa, así descrito, mediante estructuras de granularidad fina de cómputo y directivas a un ejecutable (típicamente de tipo SPMD) es un proceso automático que realiza el compilador de paralelismo de datos. Pero, hay que

tener en cuenta, que el programador tiene que entender las características esenciales de este proceso, procurando escribir código eficiente y evitando construcciones ineficientes. Por ejemplo, una elección incorrecta de algunas directivas puede provocar desbalanceos de carga o comunicaciones innecesarias. También el compilador, puede fallar en los intentos de reconocer oportunidades para la ejecución concurrente. Generalmente se puede esperar que un compilador para paralelismo de datos, sea capaz de generar código razonablemente eficiente cuando la estructura de comunicaciones sea local y regular. Programas con estructuras irregulares o comunicaciones globales pueden tener problemáticas importantes en cuanto a las prestaciones obtenidas.

En cuando a los lenguajes que soportan este modelo, podemos citar a Fortran 90, y su extensión High Performance Fortran, como el más ampliamente utilizado para implementar este tipo de paralelismo [Cha94a, Fos95]. Gracias al tratamiento de arrays, sentencias como *FORALL* y directivas *INDEPENDENT* permiten identificar la concurrencia en el algoritmo, mientras otras directivas de distribución de datos permiten especificar como serán colocados sobre los procesadores y poder proporcionar control sobre la localidad de los datos.

1.3.3. Paso de Mensajes

Este es el modelo [Kum94, Wil99] más ampliamente usado. En él los programadores organizan sus programas como una colección de tareas con variables locales privadas y la habilidad de enviar, y recibir datos entre tareas por medio del intercambio de mensajes. Definiéndose así por sus dos atributos básicos: Un espació de direcciones distribuido y soporte únicamente al paralelismo explicito.

Los mensajes pueden ser enviados vía red o usando memoria compartida si está disponible, dependiendo del modelo arquitectural del sistema (o de la virtualización software empleada). Las comunicaciones entre dos tareas ocurren a dos bandas, donde los dos participantes tienen que invocar una operación. Podemos denominar a estas comunicaciones como operaciones cooperativas ya que deben ser realizadas por cada proceso, el que tiene los datos y el proceso que quiere acceder a los datos. También, en algunas implementaciones [For97, Jia04], pueden existir comunicaciones de tipo (*one-sided*), si es solo un proceso el que invoca la operación, colocando todos los parámetros necesarios, y la sincronización se hace de forma implícita .

Como ventajas, el paso de mensajes permite un enlace con el hardware existente, ya que se corresponde bien con arquitecturas que tengan una serie de procesadores conectadas por una red de comunicaciones (ya sea interconexión interna, o red cableada). En cuanto a la funcionalidad, incluye una mayor expresión disponible para los algoritmos paralelos, proporcionando control no habitual en el paralelismo de datos, o en modelos basados en paralelismo implícito por compilador paralelizante. En cuanto a prestaciones, especialmente en las CPU modernas, el manejo de la jerarquía de memoria es un punto a tener en cuenta; en el caso del paso de mensajes deja al programador la capacidad de tener un control explícito sobre la

localidad de los datos.

Por contra, el principal problema del paso de mensajes es la responsabilidad que el modelo hace recaer en el programador. Él debe explícitamente implementar el esquema de distribución de datos, las comunicaciones entre tareas, y su sincronización. En estos casos su responsabilidad es evitar las dependencias de datos, evitar deadlocks y condiciones de carrera en las comunicaciones, así como implementar mecanismos de tolerancia a fallos [Blo00, Afr99] para sus aplicaciones.

Como en el paso de mensajes disponemos de paralelismo explicito, es el programador quien de forma directa controla el flujo de las operaciones y los datos. El medio más usado para implementar este modelo de programación es a través de una librería, que implementa la API de primitivas habitual en entornos de pase de mensajes.

PVM [Gei94] y MPI [Gro96, Don96] constituyen ejemplos de librerías que soportan el modelo de paso de mensajes. Sin embargo, hay que establecer ciertas diferencias, ya que PVM de por sí es un entorno completo que incorpora la librería y un conjunto de utilidades de sistema de soporte a la ejecución de programas por paso de mensajes, mientras MPI es una especificación, que los entornos que la implementen deben cumplir, dejando aparte (o a decisión del fabricante) como implementar el entorno de soporte para la ejecución.

Esta API de paso de mensajes incluye habitualmente:

- Primitivas de paso de mensajes punto a punto. Desde las típicas operaciones de envío y recepción (*send* y *receive*), hasta variaciones especializadas de éstas.
- Primitivas de sincronización. La más habitual es la primitiva de Barrier, que implementa un bloqueo a todas (o parte) de las tareas de la aplicación paralela.
- Primitivas de comunicaciones colectivas. Donde varias tareas participan en un intercambio de mensajes entre ellas.
- Creación estática (y/o dinámica) de grupos de tareas dentro de la aplicación, para restringir el ámbito de aplicación de algunas primitivas, permitiendo separar conceptualmente unas interacciones de otras dentro de la aplicación.

En cuanto al software que implemente el paso de mensajes, habitualmente posee las siguientes características:

- Se utilizan funciones de librería y primitivas sobre lenguajes de propósito general, siendo los más habituales Fortran o C.
- Los paradigmas (ver sección 1.4) habituales son SPMD, o bien con creación dinámica de tareas. Generalmente se soporta MIMD.
- Las comunicaciones son siempre explícitas, vía enviar y recibir mensajes, mediante las primitivas de la librería.

Así, podemos resumir las ventajas generales del paso de mensajes en:

- Versatilidad: El modelo se desarrolla sobre una serie de procesadores separados conectados por una red de comunicaciones (más o menos rápida). Este tipo de hardware se encuentra disponible en la mayoría de los computadores paralelos, y en los clusters, en los modelos arquitecturales MIMD. Si la máquina da soporte hardware extra (por ejemplo soporte de mensajes, o routing de éstos), el modelo es flexible para adaptar el hardware y obtener mejores prestaciones. También pueden adaptarse máquinas de tipo SMP o MIMD de memoria compartida, donde el paso de mensajes puede implementarse mediante mecanismos de mensajes compartidos colocados en zonas de memoria compartida.

- Completitud: El paso de mensajes es un modelo muy útil y completo, para expresar algoritmos paralelos [Gro96].

- Prestaciones: El paso de mensajes ofrece al programador el camino para expresar explícitamente la asociación de datos con procesos, aumentando así el control de la localidad, y permitiendo a compiladores, y caches optimizar su rendimiento.

Así podemos resumir que el paso de mensajes como paradigma de programación, requiere que el paralelismo se exprese de forma explícita por el programador, responsable de analizar su algoritmo/aplicación secuencial (si parte de ella) para identificar vías por las cuáles pueda descomponer la computación y extraer concurrencia. Como resultado, la programación mediante el paradigma de paso de mensajes, suele ser una actividad dura, consumidora de tiempo, y intelectualmente de alta demanda. De otro lado, las aplicaciones diseñadas por paso de mensajes tienden a conseguir buenas prestaciones, y a mantenerlas cuando el sistema aumenta a números grandes de tareas y/o procesadores.

Entre otros, algunos de los sistemas basados en paso de mensajes [Gei94, McB94, Gro96, Zav00] son PVM, MPI, BSP, Parmacs, P4, y Express. Nos detendremos en particular en PVM y MPI que son objeto de nuestro estudio de prestaciones.

Para iniciar este estudio es necesario definir una serie de conceptos relativos al paso de mensajes:

Buffer de aplicación Espacio de dirección en el programa del usuario que contiene los datos recibidos o enviados. También denominado buffer de usuario.

Buffer de sistema Espacio de dirección en el sistema (normalmente asociada a la librería de paso de mensajes) para almacenar los mensajes pendientes ya iniciados. Este espacio normalmente no es visible al programador. Dependiendo del tipo de comunicaciones será necesaria la copia entre los buffers de usuario y de sistema. El uso principal de este tipo de buffers es habilitar las comunicaciones asíncronas.

Buffering Copia temporal de los mensajes realizada por el sistema como parte de los protocolos de transmisión usados. La copia se realiza entre el espacio de los buffer de usuario (definida generalmente por el proceso), y el espacio de buffers del sistema (definido por la librería).

Comunicación bloqueante Una rutina de comunicación es bloqueante si su finalización depende de que sucedan ciertos eventos. Para operaciones de envío, los datos tienen que haber sido enviados, o bien copiados a salvo, de manera que el buffer que contenía los datos utilizados pueda ser rehusado. Para operaciones de recepción, los datos tienen que estar copiados a salvo en el buffer de recepción de manera que puedan ser usados.

Comunicación no bloqueante En el caso de que la rutina de comunicaciones vuelva sin esperar a que hayan sucedido los eventos que indiquen la finalización (como la copia de datos de espacio de sistema a usuario, o la llegada del mensaje). En este caso, no es segura la reutilización o modificación de los buffer empleados, y es responsabilidad del programador asegurar cuando el buffer de usuario podrá reutilizarse. Las comunicaciones no bloqueantes son utilizadas principalmente para permitir el solapamiento de computación y comunicación como medida de ganancia de prestaciones. Sin embargo, es necesario realizar un control adicional por parte del programador.

Comunicación síncrona Comunicación de envío que no retorna hasta que la operación de recepción correspondiente ha sido iniciada en el proceso de destino del mensaje. Normalmente mediante algún tipo de confirmación del receptor.

Comunicación asíncrona Comunicación, en que ni el emisor ni el destinatario han puesto ninguna restricción en términos de finalización.

Comunicación Ready Operación de envío donde se garantiza que el receptor estará esperando. Es responsabilidad del programador asegurar que sea así.

Envoltura de mensaje El mensaje, a transmitir, consiste normalmente en los datos, más una porción de envoltura, consistente (habitualmente) en una identificación del mensaje, del canal usado (o grupo de tareas involucrado), así como el emisor y destinatario del mensaje, más la longitud del mensaje, y posiblemente alguna información adicional dependiendo de la comunicación.

Veamos a continuación algunas de las características de las librerías más usadas de paso de mensajes.

1.3.3.1. PVM

PVM [Gei94] (Parallel Virtual Machine) es un entorno de paso de mensajes desarrollado en laboratorio de Oak Ridge, que proporciona mecanismos para la creación e intercomunicación de tareas en múltiples arquitecturas.

Algunos puntos destacados de PVM, son la inclusión de la idea de sistema paralelo con control centralizado, que incorpora unos daemons de comunicación en los nodos y un consola de manejo centralizada.

Por lo que respecta a la librería, es especialmente destacable las capacidades de creación dinámica de procesos.

Desde el punto de vista de la estructura, podemos ver al entorno de PVM como un software que nos permite que un conjunto (posiblemente) heterogéneo de computadoras (ya sean con arquitecturas serie, paralelas o vectoriales) sea visto como un único computador paralelo de memoria distribuida. PVM proporciona las funciones necesarias para crear, comunicar y sincronizar tareas sobre está máquina virtual, de manera que cooperen en la resolución de problemas. PVM introduce en lenguajes, como Fortran y C, las primitivas necesarias para la gestión de procesos, la comunicación y sincronización de estos. Las aplicaciones escritas utilizando las primitivas de la librería PVM, pueden ejecutarse en entornos MIMD, bajo paso de mensajes.

El entorno PVM se compone de dos partes principales (más una tercera opcional):

- El demonio pvm3d: Que reside en todos los computadores que componen la máquina virtual. Antes de poner en marcha cualquier aplicación PVM, hace falta iniciar este demonio de forma que el computador pase a pertenecer a la maquina virtual.

- La librería de primitivas libpvm3: contiene todas las rutinas de paso de mensajes, creación de procesos, configuración, etc., que pueden ser llamadas desde las aplicaciones de usuario.

- La consola PVM: es una tarea que permite al usuario, de una manera interactiva, inicializar, testear y modificar la configuración de la maquina virtual, así como lanzar y monitorizar aplicaciones.

En la figura 1.2 podemos ver el modelo de computación de PVM, donde una serie de procesos se comunican entre ellos, comunicación gestionada por los demonios PVM.

En la figura 1.3 vemos los diferentes niveles de comunicación que se establecen en un sistema PVM.

En cuanto las primitivas soportadas por la librería PVM, las podríamos agrupar en los siguientes grupos:

- Gestión de procesos: creación, finalización, estado, obtención identificador, etc.

- Comunicaciones: Enviar, recibir (bloqueante o no), broadcast, multicast.

- Empaquetar, y desempaquetar datos de los mensajes y gestión de buffers.

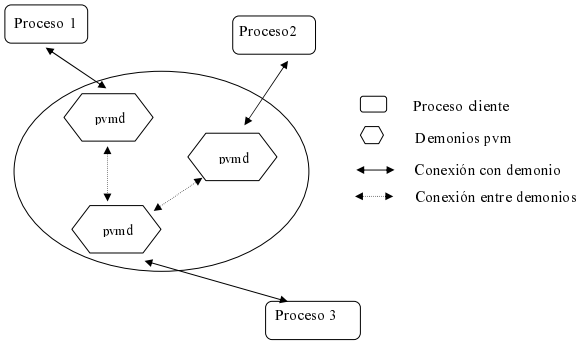

Figura 1.2: Modelo de computación PVM

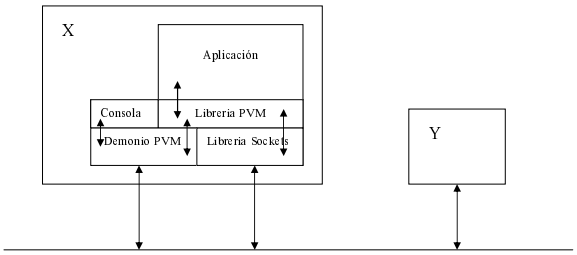

Figura 1.3: Niveles de comunicación en PVM

- Información y gestión dinámica de la máquina virtual, incluyendo tolerancia a fallos.
- Grupos de procesos dinámicos: juntarse o dejar el grupo, operaciones diversas sobre el grupo: sincronización por barreras, comunicaciones multicast, scatter, gather y operaciones de reducción.

1.3.3.2. MPI

MPI [Gro96, Fos95] es una especificación, definida originalmente en el Laboratorio Nacional de Argone, para estandarizar la implementación de librerías de paso de mensajes. Actualmente hay diferentes implementaciones, tanto libres como comerciales, en forma de librería (y normalmente más el entorno propio de ejecución) para plataformas tales como Intel, SGI, HP, Cray, etc ...

El objetivo de la especificación MPI, es demostrar que los usuarios no necesitan realizar un compromiso entre eficiencia y portabilidad. MPI permite escribir programas portables que puedan obtener ventajas del hardware y software especializado de los fabricantes.

El estándar MPI ha pasado por varias fases, desde su versión 1.0 inicial (1994), hasta su versión estable 1.1 (en 1995) [For95], con algunos refinamientos y correc-

ciones. En 1997 apareció la 1.2 y la versión 2.0, integradas las dos en la especificación MPI 2.0 [For97]. En el desarrollo de la última, la idea era incluir nuevas funcionalidades, como por ejemplo la creación dinámica de procesos. Hoy en día, la mayoría de versiones son compatibles MPI 1.1/1.2, en algunos casos hay alguna implementación parcial de la MPI 2.0, como por ejemplo LAM [Uni05], pero no totalmente desarrollada, aunque hay algún intento en este sentido en desarrollo, como OpenMPI [Pro05], que ya dispone de algunas betas de una implementación casi completa de los estándares MPI 1.x y 2.0.

En la realización de los estándar MPI, se propusieron como objetivos [For95]:

- Diseñar una API genérica
- Establecer un eficiente sistema de comunicaciones, evitando la copia de datos de memoria a memoria, favoreciendo que la computación y las comunicaciones de un proceso se realicen concurrentemente.
- Favorecer la ejecución en entornos heterogéneos (donde puedan coexistir diferentes plataformas y sistemas operativos) de programas paralelos basados en alguna implementación MPI.
- Conseguir un sistema de comunicaciones fiable, en el cual el usuario no tenga que tratar los fallos y errores de comunicaciones, sino que sea el propio sistema quien los solucione.
- Diseñar enlaces para interfaz con los lenguajes C Y Fortran.
- Diseñar la interfaz de manera que no sea muy diferente a otras existentes tales como PVM, NX, Express, P4, para que los usuarios de estas pudieran migrar rápidamente a MPI.
- Favorecer la portabilidad, para que la interfaz pueda ser implementada en la mayoría de sistemas paralelos (tanto sistemas multiprocesador o clusters de estaciones de trabajo), sin ofrecer cambios importantes tanto en el software como en el sistema de comunicaciones.
- La interfaz tiene que ser independiente del lenguaje en que va ser implementada.
- La interfaz debe ser diseñada para ser thread-safe (soporte de ejecución de threads)

Todos estos objetivos iniciales se han ido consiguiendo a través de un diseño activo, por el cual, periódicamente a través del MPI Forum, se proponen y consensúan nuevas revisiones y actualizaciones al estándar MPI, adaptándose a las necesidades de los usuarios, así como a la nueva aparición de arquitecturas y sistemas paralelos. De estas revisiones y actualizaciones han surgido las dos grandes familias de versiones de MPI: la MPI1.1 y la MPI2.

En las especificaciones de MPI 1.1 podemos encontrar:

- Comunicaciones punto a punto: Mecanismo básico de comunicaciones usado por MPI, consiste en enviar y recibir mensajes entre procesos (operaciones primitivas de paso de mensajes *send* y *receive*).
- Comunicaciones colectivas: conjunto de operaciones de comunicación que involucran a varios procesos, tales como barreras de sincronización, envíos de tipo broadcast a todos los miembros de un determinado grupo, recoger o dispersar datos de todos los procesos a uno o de uno a todos, etc.
- Creación de grupos (estáticos) de tareas: los grupos describen una colección ordenada de procesos. Cada proceso se identifica a través de un identificador de rango propio, que lo identifica para las comunicaciones entre las tareas del grupo. Estos grupos son usados para las comunicaciones punto a punto y en las colectivas.
- Creación de contextos de comunicación: A través de los contextos se separa el espacio de comunicación, con ello conseguimos que un mensaje enviado a los procesos que forman parte de un contexto no sea recibido por los que no forman parte de este.
- Creación de topologías de tareas: La topología nos proporciona un mecanismo para dar el nombre e identificar adecuadamente a los diferentes procesos del grupo, y reflejar la topología lógica de sus comunicaciones. También puede ayudar en tiempo de ejecución, a realizar el mapeado de los procesos en memoria.
- Creación de enlaces para los lenguajes C y Fortran: con ellos, se especifica la sintaxis para las versiones C y Fortran de las funciones MPI.
- Control del entorno y peticiones de información: a través de algunas primitivas, el usuario puede solicitar o cambiar el valor de ciertos parámetros de MPI, y adaptar la ejecución del entorno a las necesidades de su aplicación.
- interfaz de profiling (PMPI): a través de esta interfaz, se ofrece un modelo fácil para dar nombres a las diferentes primitivas MPI, de manera que se pueda utilizar como mecanismo para implementar aproximaciones de *wrappers* de llamadas a funciones MPI, pudiendo así sustituirlas por otras que permitan tomar medidas o trazar estas llamadas. Esto facilita la implementación de herramientas de rendimiento de prestaciones sin tener que acceder al código de la implementación de MPI

En la nueva versión MPI2 se ha añadido nuevo soporte para algunos de los objetivos iniciales que no estuvieron cubiertos con las especificaciones MPI1.1:

- Creación y gestión de tareas: se incorporo creación dinámica de tareas. Esto facilita la migración de aplicaciones PVM que tuvieran este modelo. Y también permite incorporar nuevos tipos de aplicaciones, como aplicaciones serie con parte paralela a lo largo de su ejecución.

- Comunicaciones *one-side*: con la especificación RMA (Remote Memory Access) se extienden algunas de las facilidades de comunicación, permitiendo a las tareas seleccionar parámetros de comunicación tanto en emisor como en el receptor. Así una comunicación queda dividida en dos partes, la primera es la comunicación de datos, y la segunda la sincronización de la comunicación. Para cada fase existen una serie de primitivas. El objetivo es aprovechar los recursos propios de cada arquitectura o sistema para adecuar las comunicaciones a sus recursos.
- Extensiones para operaciones colectivas.
- Interfases externas: Se añaden mecanismos para que los usuarios puedan controlar (y ampliar) MPI desde un nivel superior. Se ofrecen: control de nombres (de tipos, estructuras) personalizado; creación de operaciones de comunicación; definición de tipos de errores.
- Entrada y Salida: proporciona primitivas portables y eficientes para un sistema paralelo de E/S, soportando: particiones de datos, transmisión de ellos entre tareas hacia ficheros remotos, operaciones asíncronas de E/S.
- Nuevos enlaces a lenguajes: se añade soporte para C++ y Fortran 90.

En la librería de paso de mensajes de MPI cabe destacar, las grandes posibilidades de comunicaciones mediante primitivas colectivas, en particular la sofisticación de la flexibilidad de envió (en el proceso de partición de datos y posible tareas destinatarias); Y también destacar la flexibilidad en el paso de mensajes asíncronos en las primitivas punto a punto a tal efecto.

1.4. Paradigmas de programación

Los paradigmas de programación paralelos son una clase de algoritmos que solucionan diferentes problemas bajo las mismas estructuras de control [Han93]. Normalmente bajo la idea de paradigma se encapsula un determinada forma de realizar el control y las comunicaciones de la aplicación.

Hay diferentes clasificaciones de paradigmas de programación, pero un subconjunto habitual [Buy99] en todas ellas es:

1.4.1. Master-Worker

O también denominado Task-farming, consiste en dos tipos de entidades, el master y los múltiples esclavos. El master es responsable de descomponer el problema a computar en diferentes trabajos más simples (o en subconjuntos de datos), los cuáles distribuye sobre los workers y finalmente recolecta los resultados parciales para componer el resultado final de la computación. Los workers ejecutan un ciclo muy simple: recibir un mensaje con el trabajo, lo procesan y envían el resultado parcial de vuelta al master. Usualmente solo hay comunicación entre el master

y los workers. Uno de los parámetros importantes de este modelo es el número de workers utilizados.

Normalmente este modelo se puede ampliar con una generalización del trabajo a resolver, como una serie de ciclos donde se repite el esquema básico, El master dispone por cada ciclo de una cantidad de trabajo por realizar, que envía a los esclavos, estos reciben, procesan y envían de vuelta al master.

Este paradigma en particular posee ciertas características: como son un control centralizado (en el master), además suelen darse ciclos bastante semejantes de ejecución, lo que permite prever el comportamiento futuro en posteriores ciclos. Además se pueden tomar medidas eficaces de rendimiento observando los nodos, las cantidades de trabajo enviadas, y los tiempos correspondientes al proceso realizado en los workers, determinando cuáles son más eficientes, o por ejemplo en entornos heterogéneos que nodos están más o menos cargados de forma dinámica, y así adaptarnos de forma dinámica a estas situaciones con diferentes técnicas [Ban04].

1.4.2. SPMD

En el Single Program Multiple Data (SPMD), se crean un número de tareas idénticas fijas al inicio de la aplicación, y no se permite la creación o destrucción de tareas durante la ejecución. Cada tarea ejecuta el mismo código, pero con diferentes datos. Normalmente esto significa partir los datos de la aplicación entre los nodos (o máquinas) disponibles en el sistema que participan en la computación. Este tipo de paralelismo también es conocido como paralelismo geométrico, descomposición por dominios o paralelismo de datos.

1.4.3. Pipelining

Este paradigma está basado en un aproximación por descomposición funcional, donde cada tarea se tiene que completar antes de comenzar la siguiente, en sucesión. Cada proceso corresponde a una etapa del pipeline, y es responsable de una tarea particular. Todos los procesos son ejecutados concurrentemente y el flujo de datos se realiza a través de las diferentes fases del pipeline, de manera que la salida de una fase es la entrada de la siguiente. Uno de los parámetros importantes aquí es el número de fases.

1.4.4. Divide and Conquer

El problema de computación es dividido en una serie de subproblemas. Cada uno de estos subproblemas se soluciona independientemente y sus resultados son combinados para producir el resultado final. Si el subproblema consiste en instancias más pequeñas del problema original, se puede entonces, realizar una descomposición recursiva hasta que no puedan subdividirse más. Normalmente en este paradigma son necesarias operaciones de partición (split o fork), computación,

y unión (join). Normalmente trabajamos con dos parámetros, la anchura del árbol, el grado de cada nodo, y la altura del árbol, el grado de recursión.

1.4.5. Paralelismo especulativo

Si no puede obtenerse paralelismo con alguno de los paradigmas anteriores. Se pueden realizar algunos intentos de paralelización de cómputo, en base a escoger diferentes algoritmos para resolver el mismo problema. El primero en obtener la solución final es el que se escoge entre las diferentes opciones probadas. Otra posibilidad, es si el problema tiene dependencias de datos complejas, puede intentarse ejecutar de forma optimista, con una ejecución especulativa de la aplicación, con posibles vueltas atras, si aparecen problemas de incoherencias o dependencias no tenidas en cuenta o previstas, que obliguen a deshacer cómputo realizado previamente si algo falla. En caso que acertemos, o no aparezcan problemas, podemos incrementar de forma significativa el rendimiento.

1.5. Prestaciones y métricas

Uno de los principales problemas en el cómputo de altas prestaciones es conseguir aplicaciones paralelas (y/o distribuidas) eficientes, las cuáles sean capaces de aprovechar, de la mejor manera, los recursos que ofrecen los sistemas.

El concepto de prestaciones de una aplicación paralela es un termino complejo de definir y en el que hay que tener en cuenta múltiples facetas [Fos95, Gro99].

Tenemos que considerar además de los tiempos de ejecución y la escalabilidad del cómputo utilizado, los mecanismos por los cuáles los datos son generados, almacenados, transmitidos por red, transferidos de/a disco, y fluyen por las diferentes etapas de computación. Tenemos también que considerar los costes implicados, respecto a las prestaciones, en todas las fases del ciclo del desarrollo del software de aplicación, ya sea su diseño, implementación, ejecución y mantenimiento.

Las métricas utilizadas para medir las prestaciones serán variadas, como el tiempo de ejecución, la eficiencia paralela, los requerimientos de memoria, throughput, latencia, ratios de E/S, throughput de red, costes de diseño, costes de implementación, costes de verificación, reusabilidad, requerimientos de hardware, costes del hardware, costes de mantenimiento, portabilidad y escalabilidad. La relativa importancia que le demos a cada una de estas métricas, dependerá del problema a tratar [Hel96]. Una especificación (de requerimientos) de prestaciones esperadas, puede dar números para algunas métricas, requerir que algunas sean optimizadas por debajo de algunos umbrales, y que otras métricas sean simplemente ignoradas.

El proceso de investigar el comportamiento de una aplicación, estudiar sus prestaciones, y encontrar las causas que limitan estas, es conocido como análisis de prestaciones. En el proceso identificamos aquellas partes de comportamiento ineficiente, explorando las razones que lo causan, y cuantificamos la importancia sobre

las prestaciones globales. La información conseguida a través de este proceso tiene que proporcionarnos acciones que podamos tomar para optimizar y sintonizar la aplicación.

Con los índices de medida de prestaciones [Jai91], estamos describiendo que tipo de resultados vamos a obtener de la actividad de optimización que realizaremos. Los posibles índices de prestaciones acaban midiendo como han influido las tareas de optimización realizadas frente (a uno o más) de los siguientes aspectos:

- Tiempo de completar la aplicación: Tiempo desde que la aplicación comienza hasta que acaba (a veces también denominado *wall clock time*).
- Troughput: Cantidad de trabajos (entornos batch), o de peticiones (entornos interactivos) que el sistema puede servir por unidad de tiempo. Usándose en general como medida de la productividad.
- Eficiencia: El ratio del máximo troughput alcanzado respecto la capacidad nominal del sistema. Este representa como de bien se están usando los recursos disponibles. Si hay recursos desperdiciados entonces la eficiencia sera baja. Podemos tener una visión global, para toda la aplicación o sistema de cómputo, o una visión local analizando la tarea o el nodo concreto del sistema. En este caso también podemos observar la utilización de un sistema como la fracción de tiempo en que el recurso está ocupado sirviendo peticiones. El periodo durante el cual el recurso no es usado en trabajo útil es denominado tiempo ocioso (idle time).
- Tiempo medio de respuesta: Corresponde a la cantidad de tiempo que el usuario ha de esperar hasta que comience a recibir salida o resultados de la aplicación. En general, de forma simple, podemos presentarlo como el tiempo en la emisión de una petición por parte del usuario hasta la respuesta del sistema. Aunque, hay que tener en cuenta que tanto la petición como la respuesta no son inmediatos, y por tanto existirán tanto existirán unos tiempos de submisión de una petición, y tiempos de reacción por parte del sistema, que tendrán que ser tenidos en cuenta.
- Otros: como overheads asociados al sistema operativo o a los mecanismos de comunicación asociados, o índices de uso de recursos.

Los índices de prestaciones escogidos, para maximizar (minimizar, o por contra obtener valores medios), dependen de si queremos promocionar las prestaciones de las aplicaciones, o bien las prestaciones totales del sistema (o de alguna de sus múltiples capas software o hardware). Objetivos que pueden entrar en conflicto entre ellos.

Además cabe tener en cuenta, que debido a la complejidad con que nos enfrentamos, no hay una simple medida de prestaciones que nos pueda dar un índice único de prestaciones. Diferentes índices son necesarios para medir diferentes aspectos. Algunos de estos índices globales [Akl89, Kar90, Mol93, Alm94, Kum94, Fos95] pueden ser los que veremos a continuación.

1.5.1. Ratio de ejecución

Este índice mide la salida de resultados por unidad de tiempo. Dependiendo de como definamos la salida de la máquina podemos definir diferentes medidas de ejecución. Algunas de las más utilizadas son los MIPS y los FLOPS, como medidas de millones de operaciones en forma de instrucciones o de operaciones matemáticas en punto flotante. Esta ultima en particular es utilizada para medir las prestaciones de las máquinas de altas prestaciones construidas, como por ejemplo las listas del TOP500 [Top05]. Normalmente se plantean una serie de benchmarks estándar, en forma de un conjunto de aplicaciones (o kernels de las mismas), que se utilizan para medir el rendimiento obtenido por el sistema paralelo.

Respecto la ejecución de la aplicación paralela hay que tener en cuenta el concepto de tiempo de ejecución, que en el caso paralelo, se define desde que la computación paralela comienza hasta que el ultimo procesador finaliza la ejecución.

1.5.2. Speedup

Cuando evaluamos un sistema paralelo, estamos obviamente interesados en conocer que ganancias estamos obteniendo en la paralelización de una aplicación respecto a su implementación secuencial. Siendo esta, habitualmente, una de las razones básicas para abrazar el cómputo paralelo para aplicaciones ya existentes en versión secuencial.

El índice de speedup está definido en una computación paralela utilizando p procesadores como el cociente entre el tiempo para realizar una computación, mediante el mejor algoritmo secuencial disponible, en un procesador, entre el tiempo de realizar la computación paralela en p procesadores.

$$speedup\,(p) = \frac{T_{serie}}{T_{paralela}\,(p)}$$

En otras palabras, el speedup es el cociente entre el tiempo del procesamiento secuencial, respecto del tiempo del procesamiento paralelo. El speedup nos muestra la relativa ganancia de solucionar el problema mediante una computación paralela.

El factor de speedup es normalmente menor que el número de procesadores (p) debido al tiempo perdido por sincronización, a los tiempos de comunicación, y a otros overheads necesarios para el cómputo paralelo. Idealmente estaríamos hablando de un speedup lineal con el número de procesadores, aunque en la práctica estaríamos (siempre) por debajo de el siendo el speedup lineal el limite teórico para el que observemos en la aplicación.

Aunque cabe destacar que en algunas arquitecturas paralelas modernas, con jerarquías de memoria basadas en sistemas de cache de alto rendimiento, pueden llegar a observarse efectos de speedup por encima del lineal [Hel89]. Siendo denominado speedup superlineal, producido por los efectos de la disposición en memoria de los datos, y de los diferentes algoritmos de mapping y de predicción usados en las cache, y en los otros niveles de la jerarquía de memoria.

1.5.3. Leyes de Amdahl y Gustafson

La ley de Amdahl [Gus88] en particular pone como limite del speedup, no el número de procesadores, sino la fracción serie (α) que aparezca en la aplicación paralela [Kar90, Hor01]. En definitiva, pone en evidencia que cualquier aplicación paralela tiene una componente secuencial, que puede eventualmente limitar el speedup que podría conseguirse sobre la computadora paralela.

$$speedup\,(p) = \frac{p}{1 + (p-1)\,\alpha}$$

Estudios posteriores, como la ley de Gustafson [Gus88, Gus88a], llevaron a considerar que la ley de Amdahl daba un limite para el descenso de tiempo en sistemas paralelos dependiendo de la fracción serie, pero esto por contra podía aprovecharse para aumentar el volumen del problema manteniendo el mismo tiempo serie. O sea el volumen de problema realizado, aumentaba la fracción paralela, disminuyendo el influjo de la serie constante.

En cierta medida con estas leyes, lo que estamos midiendo es el ratio de ejecución, o la medida de la obtención de resultados por unidad de tiempo, en relación con el tamaño inicial del problema a computar. Y la relación dentro de la aplicación de los ratios de computación secuencial frente a la paralela que puedan existir.

1.5.4. Eficiencia

La eficiencia la podemos ver como el cociente entre el speedup y el número de procesadores. Nos proporciona una medida del actual grado de speedup respecto al speedup ideal (lineal). Basándose en el concepto que solo con un sistema paralelo ideal con p procesadores conseguiremos un speedup igual a p.

$$eficiencia\ paralela\,(p) = \frac{speedup\,(p)}{p}$$

En este sentido podemos ver la eficiencia como la calidad del algoritmo (o código) paralelo usado. En otro orden podemos ver la eficiencia como la fracción de tiempo en la cual el procesador es utilizado en cómputo útil para la aplicación. Pudiendo medirse de forma local a cada procesador o en global para todo el sistema.

1.5.5. Redundancia

Cociente entre el número total de operaciones ejecutada en una computación en p procesadores, y el número de operaciones necesarias para la misma computación en un sistema monoprocesador. La redundancia está relacionada con el tiempo perdido por overhead, y siempre es más grande que 1.

1.5.6. Coste

El coste de solucionar un problema en un sistema paralelo es el producto del tiempo de ejecución paralela y el nombre de procesadores utilizados. Mientras el coste de una computación secuencial es su tiempo de ejecución. El coste refleja la suma del tiempo que cada procesador consume solucionando el problema. En algunos casos de implementación de algoritmos paralelos, se utiliza para su comparación el concepto de coste óptimo referido a que el coste de la solución paralela sea proporcional al de un sistema monoprocesador.

1.5.7. Escalabilidad

Otra de las ideas subyacentes a la ley de Amdahl es la escalabilidad, que relaciona la dependencia de las prestaciones con el número de procesadores y el grado de paralelismo en el problema. Una métrica para evaluar esta escalabilidad seria precisamente la ley de Gustafson [Gus88], ya que precisamente cuantifica la escalabilidad del problema en la medida que el tamaño del problema cambia.

La escalabilidad (en una aplicación paralela) nos permitirá que delante un aumento de recursos (tamaño del sistema) y datos (tamaño problema), esta mantenga un aumento de prestaciones. En el caso de la ley de Gustafson, se observa que el incremento de tamaño de un sistema, nos permite tratar un tamaño de problema mayor, en tiempo de ejecución total parecido. En este caso se puede hablar de speedup escalado, o sea como se comporta el speedup a medida que es escalado el problema. Podemos formularlo como, sea s la fracción serie del problema, y q la paralela, para p procesadores tendremos que la ley de Amdhal quedara formulada como:

$$speedup\left((p)\right) \;=\; \frac{s+q}{s+q/p}$$

En Gustafson tenemos el tiempo como constante para la computación paralela escalada, expresemosla como s+q (el mismo tiempo que la secuencial original), y teniendo en cuenta que s+q=1. Para la ejecución secuencial necesitaremos s+pq, ya que las p partes paralelas tendrán que realizarse secuencialmente. Entonces el speedup escalado quedara como:

$$speedup_s\left(n\right) \;=\; \frac{s+pq}{s+p} = \; s+pq = p+\left((1-p)\right)s$$

De esta manera se observa que el speedup escalado no ofrece una caída rápida según la fracción serie (como sugería la ley de Amdahl), siempre que se aumente el tamaño del problema, para mantener el tiempo de ejecución total.

Capítulo 2

Análisis de Prestaciones

En este capítulo veremos los conceptos de análisis de prestaciones que usaremos a lo largo del trabajo. Estudiaremos los diferentes elementos de la aproximación clásica a al análisis de prestaciones, basada en la utilización de herramientas de visualización. Examinaremos las diferentes metodologías clásicas de recolección de información, el tracing y el profiling. Así como las diferentes consideraciones de modelos y tiempos, asociados a las comunicaciones presentes en los modelos de paso de mensajes analizados. También se realiza una recopilación de algunos modelos destacables de herramientas, de las que podríamos llamar de primera generación, por la recopilación de información normalmente destinada a dar una visión gráfica final de cara al usuario. Se introduce el concepto de análisis automático, que se desarrollara en el siguiente capítulo, en contraposición a la visualización.

2.1. El proceso de análisis

En todo sistema de análisis de prestaciones, el objetivo es conocer el comportamiento de la aplicación para poder evaluarla y poder determinar las prestaciones deseadas. El proceso de investigar el comportamiento de la aplicación, y encontrar las causas del limite de las prestaciones obtenidas, normalmente incide en una segunda etapa donde se procede a la modificación del código fuente, con objetivo de optimizar y/o adaptar el programa para evitar los problemas de prestaciones. Estas dos actividades se repiten de forma iterativa hasta que la aplicación alcance las prestaciones deseadas.

Básicamente podríamos ver el proceso como una serie de fases dentro de un proceso iterativo [Pan99]:

- Identificar. ¿Hay problemas de prestaciones? ¿Qué síntomas aparecen?
- Localizar. ¿En que puntos la ejecución se observan disminuciones de las prestaciones? ¿Qué está causando el problema?

- Reparar. ¿Qué medidas se pueden tomas para evitar/eliminar el problema?
- Verificar. ¿Arreglamos el problema? Sino deshacer, y volver a Localizar.
- Validar. ¿Hay aún más problemas de prestaciones? Si, entonces volver a Identificar.

El proceso iterativo en el análisis de prestaciones anterior, no es obvio, ¿cuanto tiene que durar?. La cuestión de fondo es ¿Hasta cuando hay que buscar mejorar las prestaciones?, la solución no es clara, en muchas ocasiones se produce mediante varios ciclos (parecidos a los anteriores) mediante las hipótesis de prestaciones que determina el desarrollador, y que intenta probar, hasta que, normalmente, se queda sin más hipótesis por probar. O bien, se llega a algunos compromisos entre los diferentes parámetros de ejecución, y los índices de medida de prestaciones que se estén utilizando.

Normalmente se ha intentado sistematizar este proceso, a través de la idea de ciclo de creación de hipótesis, mediante validación de estas teniendo en cuenta la información que se disponga de los índices de prestaciones. Mediante la definición de un cierto modelo de prestaciones, y unos limites de ganancia de prestaciones que se quieran obtener, de manera que encontremos un limite práctico de esfuerzo respecto a los objetivos que pretendamos conseguir, para el análisis de prestaciones de la aplicación paralela.

Las hipótesis formuladas estarán basadas en uno o más parámetros del modelo de prestaciones, este deberá tener en cuenta todos aquellos parámetros e índices de prestaciones que queramos considerar para la mejora de nuestras prestaciones en la aplicación. Podemos por ejemplo, considerar modelos donde nos interese: a) El mínimo tiempo de ejecución global para nuestra aplicación, b) Una eficiencia óptima de los recursos (por ejemplo uso balanceado de cómputo en los procesadores), c) Una escalabilidad adecuada según el incremento de procesadores utilizados en el sistema, d) O cualquier combinación de índices (speedup, eficiencia, etc.) que nos permita obtener un compromiso adecuado entre ellos (por ejemplo speedup versus coste)

El análisis de prestaciones, respecto a las preguntas iniciales que nos hacíamos en esta sección, puede verse también como una análisis multidimensional, en varios ejes, de lo que está sucediendo en la aplicación. Cada dimensión (o eje) nos presenta una posible visión de las prestaciones, siendo posibles diferentes aproximaciones, como por ejemplo: 1) Localización: ¿Donde ocurren las ineficiencias? (relación con las posiciones del código, o el tiempo); 2) Observación: ¿Qué ocurre? ¿Cuáles son las ineficiencias?; y 3) Explicación: ¿Porqué ocurre? ¿qué lo provoca?. Varias herramientas de análisis de rendimiento, como veremos utilizan esta aproximación multidimensional (ver por ejemplo los casos de Paradyn [Irv96] y Expert [Wol03]).

En las tres secciones siguientes de este capítulo, identificaremos los diferentes componentes del análisis de prestaciones, incidiendo primero en la localización, respecto al consumo de tiempo (eje temporal), y respecto al lugar donde este es

consumido en el código (eje espacial). Analizamos después los detalles de los entornos estudiados, de paso de mensajes MPI y PVM, respecto a sus primitivas básicas, y modelos de consumo de tiempo más detallados que involucren los diferentes elementos presentes en estos entornos.

2.2. Eje temporal: ¿En qué se consume el tiempo?

En el paradigma de paso de mensajes tendremos que considerar algunos tiempos asociados a los diferentes tipos de comunicaciones, así como a las sincronizaciones y a los tiempos de CPU.

En general para cualquier computación de una aplicación, ya sea serie o paralela, podríamos separar los tiempos de ejecución en: a) tiempo de usuario y b) tiempo de sistema. Relacionado con los diferentes modos de usuario y kernel de ejecución. Es en el modo usuario donde las instrucciones generadas por el compilador son ejecutadas, así mismo como las llamadas a funciones externas enlazadas desde librerías a nuestra aplicación. En el modo kernel, se utiliza cuando se requieren el uso de servicios proporcionados por el sistema operativo (normalmente por el kernel), como por ejemplo la E/S. Así en general obtenemos el concepto de tiempo de CPU como:

$$CPU\ time\ =\ user\ time\ +\ system\ time$$

Pero hay que tener en cuenta que mientras el tiempo de usuario no es sobrecargado por nada más que su ejecución, por contra en el tiempo de sistema, debido a la multiprogramación, se ve afectado por la carga del resto del sistema, y las llamadas al sistema y/o interrupciones producidas desde otras aplicaciones. En el caso del tiempo de usuario también tenemos que tener en cuenta los efectos de la jerarquía de memoria, en la resolución de los accesos a memoria, con especial atención a los efectos de las caches, y en concreto a los fallos de cache.

También nos influye disponer de memoria virtual, por si los tamaños de aplicación o por multiprogramación, ejecutamos nuestra aplicación fuera de los limites de memoria física disponible.

Algunos de estos factores afectan al comportamiento de un tercer tiempo, el tiempo transcurrido (*elapsed time*), o tiempo que ha pasado desde que la aplicación se ha iniciado. Este tiempo si la aplicación pasa su mayor parte del tiempo realizando cómputo sera ligeramente superior al tiempo de CPU, en el caso de que aparezcan diferencias importantes, las razones son debidas a:

- La existencia de multiprogramación, y por tanto la compartición de CPU con otros programas activos.
- Un uso elevado de E/S.
- Se requiere un ratio más elevado de ancho de banda de memoria del que está disponible (en particular aplicable a multiprocesadores).

- Uso de memoria virtual o intercambio, si por tamaños de aplicación o por multiprogramación, ejecutamos nuestra aplicación fuera de los limites de memoria física disponible.

El porcentaje de utilización, idealmente cercano a 1, seria el cociente entre el tiempo de CPU respecto del tiempo transcurrido. En el caso de aparición de los factores anteriores el ratio seria inferior a 1, indicándonos el porcentaje de tiempo útil respecto a los factores externos.

Esto sería útil para la medida en el total de la aplicación, respecto a porciones de esta, necesitamos poder medir porciones de programa, para permitir aislar las prestaciones de un segmento o porción de código. La técnica básica consiste en medir mediante rutinas de tiempo, el tiempo antes de realizar la tarea X, realizar la tarea X (o sea el código interesados en medir), medir el tiempo después de completar X, y finalmente realizar la resta de tiempos para disponer del tiempo en completar X.

En estos casos siempre hay que tener el cuenta que las medidas de tiempos en X producen también cierta intromisión, y pueden afectar a factores como fallos de instrucciones en cache, o a la paginación de la memoria virtual. Por tanto se hace difícil medir con cierta precisión intervalos muy pequeños (fuertemente dependiente del hardware de tiempos disponible en la máquina y de su resolución), con lo cual hay que intentar establecer un compromiso entre lo que se quiere medir, y el tiempo extra que pueda tener el proceso de medición.

Otra aproximación a la medida de tiempos, es la usada (como veremos más adelante) por muchas de las herramientas de profiling, que es la medida de tiempos en subrutinas, para determinar cuáles son las predominantes en el tiempo de la aplicación. Básicamente se utilizan mediciones de cada subrutina, llamemos la foo, desde que entramos en la subrutina hasta que salimos de ella. Aquí hay que tener en cuenta dos aspectos, si el tiempo es inclusivo o no, respecto a contener (o no) a su vez los tiempos de las subrutinas que son a su vez llamadas por foo.

Cuando analizamos una aplicación mediante medición de tiempo de sus subrutinas, lo que obtenemos es un perfil de la aplicación en forma de distribución de porcentaje total de tiempo de aplicación respecto sus rutinas. Este perfil puede presentar dos formas [Dow93]: a) Una o unas pocas rutinas suponen el mayor porcentaje de tiempo. o b) hay una distribución más o menos uniforme del porcentaje tiempo entre las rutinas.

Esto supone en la práctica que en el caso a) la ejecución se encuentra concentrada en una o pocas rutinas, y por tanto tenemos que enfocar nuestra esfuerzos por optimizar las prestaciones en esas rutinas. Por otro lado en b), el tiempo de ejecución está distribuido de forma uniforme en las rutinas de la aplicación y por tanto, no puede obtenerse una serie de subrutinas candidatas a la optimización, y habrá que tomar estrategias globales de optimización. Éstos serían los dos casos extremos, normalmente una aplicación se movería entre uno y otro, seguramente con un número intermedio de rutinas que significasen un porcentaje más o menos elevado del tiempo total (por ejemplo un 50 o 60 %).

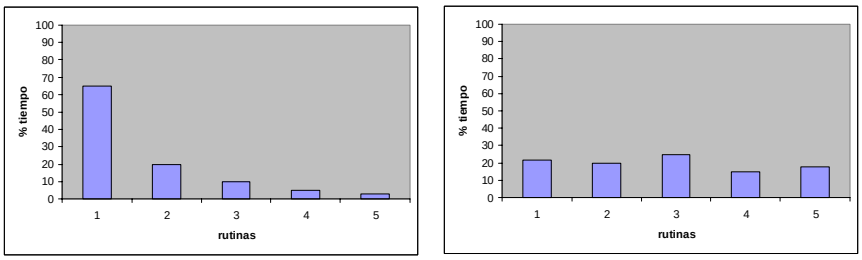

Figura 2.1: Perfiles de las rutinas en una aplicación: destacado y plano

Por otra parte en las mediciones hechas sobre rutinas también hay que tener en cuenta algunos factores:

- En el caso de que los niveles de llamadas entre las subrutinas no sean planos, tenemos que disponer de operaciones de colapsar o expandir el tiempo de una determinada subrutina, entre las que han sido llamadas por esta. Normalmente esto crea un grafo en la ejecución de la aplicación, de las subrutinas llamadas. Hay que vigilar con algunos aspectos como la recursión (crea ciclos de la misma rutina), o llamadas cruzadas entre grupos de rutinas.

- La medición sobre rutinas demasiado pequeñas puede causar una importante perturbación, debido otra vez a que el tiempo en el sistema de medida puede superar al de cómputo de la rutina.

- Funciones con cómputo muy pequeño, pueden concentrar su tiempo en el paso de parámetros que se efectúe, siendo este el valor predominante. En particular el uso de parámetros por copia (en lugar de referencia), puede incrementar este valor de forma importante.

- Lenguajes (y compiladores) que permitan funciones inline, pueden crear contradicciones con las mediciones (por ejemplo, desaparecen los tiempos asociados al paso de parámetros, al salto y al retorno de subrutina).

También hay que tener en cuenta los limites de las optimizaciones posibles, si actuamos sobre un número determinado de rutinas que supongan un porcentaje concreto, no podemos esperar obtener un beneficio mayor que el que representen esas rutinas. Por ejemplo si actuamos sobre rutinas que suponen un 50 % de la ejecución no podemos obtener beneficios mayores que una reducción del 50 % del tiempo de la aplicación (siempre que no queramos cambiar la estructura de la aplicación o los algoritmos usados). Esto nos lleva a otro punto importante, el de gastar esfuerzos de optimización allí donde realmente sea importante: no está bien guiado un proceso donde gastamos, por ejemplo, un porcentaje del 90 % del tiempo disponible para optimizar, sobre un lugar que solo supone un 10 % del tiempo de ejecución.

2.3. Eje espacial: Localización en código

Durante la ejecución de una aplicación, los caminos de llamadas, entre las rutinas de la aplicación, que se producen, pueden visualizarse bien mediante un grafo de llamadas estático, que puede ser obtenido mediante análisis del código fuente de la aplicación, definiendo que funciones existen, y en que nivel a partir del raíz (programa principal) son llamadas. De esta manera podemos determinar el grafo de quien llama a quien. En particular también es aplicable en el análisis de código, para determinar la cobertura del código, por ejemplo si hay funciones que no son llamadas en ningún momento, o desde un único o pocos puntos.

Pero hay que tener en cuenta que durante la ejecución del código, las llamadas producidas dependerán fuertemente de los datos que intervengan, y del resultado de la evaluación de las condiciones del código. En este sentido para cada ejecución de la aplicación (experimento) podemos obtener un grafo de llamadas dinámico (call graph), donde observaremos las llamadas producidas, y podremos así mismo determinar la cobertura dinámica del código [Bal96, Ber04]. Estos experimentos pueden ser acumulables, ya sea para su estudio por separado, o conjunto para determinar bajo que condiciones, y que datos se producen determinadas llamadas, o estudiar que cobertura de código de la aplicación hay en un serie determinada de ejecuciones. En particular del estudio de experimentos múltiples, respecto los grafos de llamadas, podemos determinar que uniformidad tiene la aplicación respecto a las variaciones de datos.

La información producida en un grafo de llamadas (call graph) es interesante, ya que además de indicarnos las funciones que consumen tiempo, nos indican en que contexto se produce, bajo que condiciones se cumplen, y bajo que experimento con que datos de entrada. Algunas herramientas de prestaciones usan el grafo de llamadas [Der02], junto con métricas tomadas para el análisis de prestaciones.

Los grafos de llamadas suelen producirse (como se ha visto) de forma dinámica en tiempo de ejecución, pero también es posible, crearlos a priori, mediante análisis estático del código fuente, o mediante análisis de código binario (sin ejecutar). En estos dos últimos casos se intentan determinar que llamadas y cuando aparecen, bien sea por la aparición de llamadas explicitas a funciones, o bien por código binario de salto a subrutinas (identificables por la información de depuración incluida en el ejecutable). En estos casos, frente al tiempo de ejecución, la problemática está en que no examinamos las dependencias de la formación del grafo según las condiciones y los datos de entrada que se producirán en ejecución.

Como contrapartida, se puede atacar el problema de los grafos de llamada, con una aproximación dual, primero de forma estática, por análisis de código fuente y/o imagen binaria, y segundo por el grafo generado en tiempo de ejecución. Así podremos observar las diferencias entre uno y otro, que partes se activan, cuáles no (indicando la cobertura del código original de la aplicación), y bajo que condiciones booleanas o de datos presentes.

Otro problema habitual de los grafos de llamadas, es su tamaño respecto al número de funciones y las llamadas que existan, en particular puede llegar a ser

bastante más problemático en el caso de los grafos estáticos, ya que disponen de todas las posibilidades de llamada existentes, mientras los dinámicos solo los que realmente se producen en ejecución. Se han propuesto técnicas [Knu05, Knu05a] que permiten colapsar nodos (o subgrafos) comunes, o bien estructuras recursivas presentes en el grafo global para disminuir el almacenamiento de este.

2.4. Entornos: Modelos de paso de mensajes

Como primera aproximación, en todo sistema basado en paso de mensajes, hay algunos parámetros a tener en cuenta:

- Ancho de banda: En el sistema de comunicación usado, el ancho de banda es la máxima cantidad de datos que podremos transmitir por unidad de tiempo, una vez iniciada la transmisión. Grandes anchos de banda nos proporcionan transferencias eficientes de grandes bloques de datos entre procesadores.

- Latencia: El mínimo tiempo, en un sistema de comunicaciones, para transmitir un objeto. El tiempo (como veremos), incluye tanto overheads debido a la red, como overheads software de las operaciones primitivas de envío y recepción. La latencia es un factor importante, ya que determina la granularidad de computación versus comunicaciones, y afecta al mínimo tiempo necesario para que un segmento de código podamos obtener speedup a través de la ejecución paralela. Básicamente, si un segmento de código se ejecuta en un tiempo menor que el que necesita para transmitir su resultado (tiempo afectado por la latencia), si ejecutamos ese mismo código secuencialmente en el procesador que necesita el resultado, sera más rápido que una ejecución paralela.

- Tipos de comunicaciones soportadas: además de las comunicaciones punto a punto entre procesadores, normalmente iniciadas desde un procesador origen a uno destino, se ofrecen comunicaciones sobre grupos de procesadores (o de tareas asignadas a procesadores). Podemos ofrecer comunicaciones, de un emisor a todos los posibles receptores (broadcast), de un emisor a unos receptores determinados (multicast), o sincronización por barrera (barrier) de todos, o de algunos de los procesadores (y/o tareas). Además de otras comunicaciones basadas en la repartición o recolección de datos, con patrones de partición diversos (variaciones de scatter y gather). Normalmente (como vimos en PVM y MPI), para soportar las comunicaciones [Alm94], todo sistema de paso de mensajes ofrece el concepto de canales, y primitivas de envío (send) y recepción (receive) a través de ellos. Si el canal dispone adicionalmente de cola de mensajes, el emisor después de enviar el mensaje puede continuar (no se bloquea) sin a esperar a que el receptor tenga el mensaje. Hablamos entonces, en este caso, de comunicación asíncrona (o no bloqueante). En el caso del receptor, este puede intentar recibir, y esperar a

tener el mensaje completo (bloqueante) o continuar sino tiene mensaje en su cola de recepción (no bloqueante), y volver posteriormente a comprobar la recepción.

Veremos en los siguientes apartados, con más detalle, modelizaciones del comportamiento de las operaciones características del modelo de paso de mensajes, y de los tiempos que podemos tener asociados, teniendo en cuenta los anteriores parámetros en la modelización de los tiempos.

2.4.1. Comunicaciones p2p y colectivas

En las comunicaciones punto a punto (o también denominadas a veces como *two-way communication*), como típicamente la operación de envío y recepción (*send* y *receive*), un emisor emite un mensaje que es captado por un receptor. En sus formas más simple, los prototipos de estas operaciones, serían como las siguientes:

send(sbuffer, nelem, dest)

receive(rbuffer, nelem, source)

Donde el buffer de la operación de envío guarda los datos a ser enviados, y el buffer de la operación de recepción almacena los datos recibidos. La variable nelem representa el número de unidades de datos a transmitir (ya sea por envío o por recepción). El parámetro nelem a veces no se hace explicito, ya sea por considerarse un único elemento de transmisión, o bien si los datos son de tipos predefinidos del sistema o bien por el usuario. Y los últimos parámetros representan, la correspondencia establecida con la otra tarea que participa en la comunicación punto a punto, poniendo en relación el emisor con el identificador de la tarea destino, y al receptor con el identificador de la tarea emisora.

En la mayoría de librerías de paso de mensajes, normalmente se suelen incluir algunas modificaciones a estos parámetros básicos de las operaciones, siendo normalmente utilizados parámetros extra refiriéndose a: a) Tipos de usuario, expresándolos con dos parámetros, de tipo de dato y contadores de unidades de ese tipo, o bien por contra mediante mecanismos de empaquetado previo, con funciones de empaquetado personalizadas por tipo, previas a la operación de envió, y desempaquetados posteriores a la operación de recepción. b) Los identificadores de emisores en recepción pueden ser substituidos por comodines, de forma que un receptor permita la recepción de cualquier dato que esté destinado a el, independientemente del emisor que lo haya generado. c) Por otro lado pueden introducirse conceptos de contexto de comunicación. Para no interferir con otras partes del sistema, se aisla diferentes contextos de comunicación, por ejemplo mediante el concepto de comunicador en MPI.

A partir del concepto de envío y recepción básico se construyen los diferentes modos de las primitivas disponibles (como vimos en el capítulo de introducción), además de tener en cuenta las posibilidades de introducción (o no) de buffers intermedios en el proceso de comunicación.

Respecto a estas posibilidades, teniendo en cuenta los mecanismos de buffers y las técnicas de bloqueo o no de las comunicaciones, nos encontramos con las siguientes posibilidades:

En las primitivas bloqueantes, la idea es mantener correcta la semántica de las operaciones de comunicación, para evitar que la comunicación tenga accesos no seguros a datos que pueden no estar disponibles en el momento de utilizarlos o que se pueden ver modificados durante la operación. En el caso de envió puede verse como que la primitiva se bloqueará hasta que se pueda garantizar que la semántica no se violará de vuelta de la primitiva independientemente de lo que pase a continuación en el programa. Básicamente se nos intenta garantizar la integridad de los datos usados durante el proceso de comunicación. Esto puede realizarse con dos mecanismos diferentes:

- Envío/recepción bloqueantes sin buffers: En este caso la primitiva de envío no retorna de su llamada, hasta que la correspondiente primitiva de recepción aparece durante la ejecución de la aplicación. Cuando esto ocurre se inicia la comunicación, y la primitiva de envío no retorna hasta completar la comunicación. Normalmente esto necesita un protocolo de *handshake* entre emisor y receptor. El emisor emite una petición de comunicación, el receptor la acepta, y el emisor inicia la operación. Las problemáticas de este mecanismo son varias, respecto a la posible aparición de tiempos de espera grandes, así como deadlocks entre las tareas si las dos iniciasen a la vez operaciones de envío o recepción simultaneas entre ellas.

- Envío/recepción bloqueantes mediante buffers: En muchos casos para solucionar los problemas anteriores se usan diferentes aproximaciones de buffers, ya sean predefinidos por el sistema para las comunicaciones, o habilitados por el usuario, tanto en el lado emisor como en el receptor. En estos casos, durante el envío el emisor solo se encarga de realizar la copia de los datos de la comunicación sobre el buffer, que se designe, y retorna de su llamada al terminar la copia. Así el emisor puede continuar su ejecución, sin problemas de semántica en los nodos. El receptor recogerá cuando se produzca la primitiva, sus datos del buffer. En este caso, la implementación de las primitivas depende de los mecanismos hardware existentes para las comunicaciones, dependiendo de si el sistema ofrece buffers hardware, o bien las tareas deben crearlos en el espació de usuario, y el sistema de paso mensajes en el espacio de sistema. También se pueden utilizar mecanismos por interrupción donde la aparición del envío, interrumpa al receptor, para que se copien los datos en el buffer del receptor (necesitándose así tan solo buffers a un lado de la comunicación). En este mecanismo (bloqueo con buffers) podemos ver que añadimos todo el coste de la gestión de los buffers, y el consecuente almacenamiento, si el sistema no dispone de hardware adecuado. Respecto a los deadlocks, pueden seguir produciéndose si se realizan recepciones simultaneas entre las tareas, debido al bloqueo producido en las primitivas de recepción.

En las operaciones no bloqueantes, se intentan evitar los tiempos de espera que se pueden producir en las comunicaciones, por medio de forzar al programador a usar un nivel más de control sobre la aplicación. Se traspasa el control de la semántica de las comunicaciones de forma explicita al código de la aplicación, por medio de primitivas de test y verificación de estado de las comunicaciones. El objetivo es proporcionar unos envíos y recepciones más rápidos, en la medida que el programador puede asegurar que así pueden producirse, o sea capaz de usar los tiempos que eran anteriormente de espera, en mantener un solapamiento de cómputo con las comunicaciones. En este caso, después de realizar la operación de envío o recepción, la tarea puede continuar inmediatamente, pudiendo continuar con cualquier computación que no involucre (o dependa) de completar la comunicación previa. Más adelante, puede comprobarse que se haya completado la comunicación, bien puntualmente, o mediante bucles de cómputo y comprobación, o por contra realizar una espera activa para que se complete. Los tiempos de espera de las comunicaciones bloqueantes, se convierten en posibles solapamientos de cómputo y comunicaciones. Por contra todo el proceso de comprobación ha de realizarse explicito.

En las librerías típicas de paso de mensajes, como PVM y MPI, se utilizan tanto primitivas bloqueantes como no bloqueantes. Las bloqueantes facilitan una programación más fácil y segura (en semántica de las operaciones), y las no bloqueantes son útiles para la optimización de las prestaciones, gracias a su ocultación de los overheads de comunicaciones por medio del solapamiento con el cómputo. Aunque en este último caso, tenemos que generar una programación explicita que nos evite accesos no seguros a los datos, cuando no están disponibles aún, o bien están siendo usados en la comunicación.

Algunas librerías, como PVM, presentan unas pocas primitivas genéricas, restringiéndose habitualmente solo a operaciones de envío y recepción básicas, en versión bloqueante y no bloqueante: *pvm_send* (envío bloqueante mediante buffer), *pvm_recv* (recepción bloqueante), *pvm_nrecv*(recepción no bloqueante), todas mediante el uso de buffers activos, sobre los que se empaquetan, y desempaquetan los datos de comunicación.

Respecto a la librería MPI, como ya comentamos en la introducción, uno de sus objetivos era maximizar las prestaciones de la librería, mediante adaptar lo mejor posible el estándar a la implementación física de máquina paralela que dispongamos [Ban99, Ber01]. Un hecho en particular, donde se observa esta tendencia, es en la definición en el estándar MPI de los modos de comunicación disponibles para la operación de *send* punto a punto. Como vemos en la tabla (2.1) siguiente, se definen diferentes modos de envío disponibles, que se encuentran relacionados con el protocolo inferior que se usara de send/receive. En la tabla se observan las llamadas accesibles en cada modo, y las versiones bloqueantes y no bloqueantes (primitivas con I) en cada modo.

Las primitivas colectivas, en el paso de mensajes, surgen de la necesidad de comunicación de datos, y/o compartición entre varias (o todas) las tareas participantes en la aplicación [Mar99]. En algunas librerías, el concepto de colectiva se

Modo de envío	Notas	Primitivas
Standard	No se asume que la rutina de recepción este presente, los buffers son dependientes de la implementación	MPI_Send, MPI_ISend
Buffered	El envío puede retornar aunque no este presente la recepción	MPI_Bsend, MPI_Ibsend
Synchronous	Envío y recepción pueden presentarse en cualquier momento, pero solo se completaran a la vez	MPI_Ssend, MPI_Issend
Ready	Solo puede realizarse si la recepción está ya presente	MPI_Rsend, MPI_Irsend

Tabla 2.1: Modos de envío en MPI

reduce a las tareas pertenecientes a un determinado contexto, ya sea porque: a) las tareas se encuentran en una serie de grupos estáticos o dinámicos, y el concepto de colectiva se restringe al grupo al que pertenecen; o bien b) Se integran en contextos de comunicación, de manera que las tareas solo pueden comunicarse colectivamente (o punto a punto) con las que se encuentren en su mismo contexto de comunicación. Por ejemplo:

- En PVM, las comunicaciones colectivas se implementan sobre grupos dinámicos de procesos, de manera que la operación colectiva se realiza sobre los miembros del grupo. Existe una primitiva de multicast, que permite un envío a un array de identificadores de tarea (en este caso no es necesaria la pertenencia a un grupo). Las tareas pueden pertenecer a la vez a diferentes grupos, tienen su identificador de tarea denominado tid, y su número de identificación dentro de cada grupo que formen parte.
- En MPI (versiones 1.x), los grupos de tareas son estatícos, definidos en arranque de la aplicación, pero existe el concepto de comunicador (*communicator*), como dominio de comunicación, donde la pertenencia al comunicador define la posibilidad que las tareas puedan comunicarse entre si. Existe un comunicador *MPI_COMM_WORLD* global, que incluye todas las tareas de la aplicación, pero es posible crear nuevos comunicadores que incluyan a subconjuntos de tareas. Cada tarea tiene un identificador, denominado rango (*rank*) dentro de cada comunicador al que pertenece, que le identifica (entre 0 y número de tareas - 1) en las operaciones de comunicación que se realicen con el comunicador utilizado. También es posible crear un intercomunicador, que permite comunicar a procesos presentes en diferentes comunicadores. Existe también el concepto de grupo, pero solo como lista de tareas que pertenecen a un comunicador (o intracomunicador) dado. Las operaciones colectivas se realizan sobre las tareas pertenecientes al identificador del comunicador usado en las primitivas colectivas.

Respecto a las operaciones colectivas ([Gor00, Vad01, Wor01, Vad04, Pje05]) son habituales las de: broadcast, multicast, Gather y Scatter, Reduce, All-to-All; con diversas variaciones dependiendo de las tareas participantes, y los datos que se transmitan. En la mayoría de ellas, aparece el concepto de tarea *root* que es la que suele encargarse de la gestión de los datos de la comunicación, así como de las particiones que sean necesarias generar, ya sea para enviar o recibir los datos. En todos estas primitivas suele ser habitual un concepto de bloqueo, en el cual una tarea abandona la llamada, cuando se ha completado su rol de participación, sin importar el estado de las otras tareas participantes.

En el broadcast, una tarea genera un mensaje, que es consumido de forma compartida por todas las tareas de la aplicación. Siendo el multicast una versión especializada cuando el número de tareas solo es un subconjunto de todas las presentes. En particular la primitiva de broadcast tiene el efecto de una sincronización de tareas, ya que la operación como tal no se inicia hasta que todas las tareas hayan realizado la llamada correspondiente. La primitiva de broadcast es dependiente del sistema, y en algunos casos podemos ver que algunas implementaciones de broadcast (y/o multicast) [Sup99, Kar00], pueden disponer que se realice simplemente por emisiones repetidas del mismo mensaje a los receptores (Caso de PVM), si la arquitectura del sistema paralelo no permite un flujo compartido entre varias tareas y/o nodos de computo.

La operación de Scatter, nos permite repartir datos desde la tarea root, a las restantes tareas, habitualmente datos en forma de array, que acabamos repartiendo de forma homogénea al resto de tareas. También hay primitivas que soportan particiones no uniformes de número de datos a las diferentes tareas.

Con Gather realizamos el proceso inverso de recopilación de datos, a partir de datos dispersos entre múltiples tareas, para centralizarlos en la tarea con rol de root. Otra vez podemos recibir el mismo número de datos de cada tarea o bien un número variable.

El caso de la operación de Reduce, la podemos ver como una especialización del Gather, la idea es recoger los datos para realizar un cierto cómputo que sera el que finalmente guardemos. Es típico disponer de una serie de cómputos predefinidos básicos, tipo suma, resta, producto, max, min, etc. siendo posible que el usuario pueda definir funciones de cómputo a usar con la primitiva de Reduce.

En las operaciones de tipo All, todos las tareas participantes realizan una operación con datos provenientes a su vez de todos las demas tareas.

Otra operación colectiva típica, es la sincronización de las tareas por barrera, habitualmente denominada Barrier, en ella las tareas esperan hasta que todas hallan alcanzado la llamada a la primitiva.

En la siguiente tabla 2.2, vemos las llamadas disponibles en las librerías PVM y MPI (1.x), para la realización de las operaciones colectivas.

En la librería MPI, existen algunas colectivas especializadas, en particular las que disponen del sufijo v, permiten particionamientos no uniformes (del número de elementos) de los datos entre las tareas que intervienen en la comunicación. También existen versiones especializadas de tipo All para las operaciones Gather

Operación	PVM	MPI
Broadcast	pvm_bcast	MPI_Bcast
Multicast	pvm_mcast	-
Scatter	pvm_scatter	MPI_Scatter, MPI_Scatterv
Gather	pvm_gather	MPI_Gather, MPI_Gatherv, MPI_Allgather, MPI_Allgatherv
Reduce	pvm_reduce	MPI_Reduce, MPI_Reduce_scatter, MPI_Scan, MPI_AllReduce
All	-	MPI_Alltoall, MPI_Alltoallv, MPI_Allgather, MPI_Allgatherv, MPI_Allreduce
Barrier	pvm_barrier	MPI_Barrier

Tabla 2.2: Operaciones colectivas en las librerías PVM y MPI

y Reduce, que permiten una operación de ese tipo, de manera que reciban el resultado final todas las tareas participantes, y no solo la tarea root. También existen operaciones especializadas que combinan más de un tipo, como Reduce_scatter, que integra una operación Reduce con el posterior envío de los resultados (scatter) a todas las tareas.

Respecto a los mecanismos de contexto utilizados, en PVM se usan colectivas sobre grupos dinámicos de tareas, a excepción de la operación de multicast que está implementada con envíos sobre array de identificadores de tareas (tids). En el caso de MPI, se utilizan los grupos estáticos que formen parte de un determinado contexto de comunicación expresado por el comunicador (*communicator*) usado. En todas las primitivas hay que definir la tarea que actuara de root, a excepción del tipo All, donde todas las tareas son a su vez participantes y root.

A parte de los diferentes mecanismos de comunicaciones utilizados por las librerías de mensajes, también hay que tener en cuenta algunos puntos más de categorías de primitivas presentes:

- Creación de tareas: Normalmente mediante bien técnicas estáticas o bien por creación dinámica. En el caso estático, todos las tareas son especificadas antes de la ejecución, y el sistema ejecutará un número fijo de tareas. Normalmente en este caso se utilizan aproximaciones mediante linea de comandos que le dan al sistema de ejecución, que tareas es necesario arrancar, donde, y usando que recursos. Ya sea mediante una especificación de tareas

diferentes en un paradigma de tipo master y workers, o bien mediante tareas iguales en un paradigma SPMD, donde se mezclan los códigos, y según el identificador de la tarea en tiempo de ejecución, se activa uno u otro código, seleccionando diferentes partes del código de la tarea. Por otra parte, en una creación dinámica de tareas, estas pueden crearse, ponerse en ejecución (y destruirse) por parte de otras tareas ya existentes. Una primitiva típica en este caso es `spawn(numtareas)` que inicia varias nuevas copias de una tarea predefinida (o bien dada por parámetro a partir de su ejecutable), primitiva disponible, por ejemplo en la libreria PVM.

- Información: Diversas primitivas, no directamente relacionadas con las comunicaciones, permiten obtener información de la misma tarea, de las tareas participantes en la aplicación o del entorno de ejecución (de la máquina virtual).

- Configuración: Primitivas que permiten optimizar comportamientos, a traves de ajustar diversos parámetros configurables de la libreria. También en algunos casos, se permite alterar el entorno de ejecución de forma dinámica, al agregar o quitar nodos de ejecución.

2.4.2. Análisis de tiempos en paso de mensajes

En los entornos de paso de mensaje, con las comunicaciones explicitas tendremos que tener en cuenta diversos factores que nos afectaran:

- Tiempo de startup de la comunicación, teniendo en cuenta los sistemas de buffers de usuario y/o subsistemas que haga falta preparar, posibles empaquetamientos, y codificaciones en el caso de entornos heterogéneos. En muchos casos este tiempo es superior al tiempo de un pequeño envío. Normalmente es mejor enviar datos en paquetes grandes, que muchos paquetes pequeños (en los que influirán mucho más los tiempos anteriores), siempre que las condiciones de red, y trabajo lo permitan.

- Latencia de la red: desde que emisor comienza la transmisión, hasta que un receptor (preparado) capta la transmisión. También puede verse como el tiempo para enviar un mensaje de 0 bytes.

- Ancho de banda: capacidad de bytes/segundo para enviar por el canal de comunicación. Disponible entre las tareas, o bien disponible globalmente para todo el sistema.

- Contención y congestión de la red, dependiendo de la red de interconexión, su capacidad y la existencia de único o varios caminos de comunicación, así mismo como técnicas de ruteo. La contención puede reducirse con una distribución uniforme de mensajes en la red disponible.

- Dedicación de los canales, o compartición.

Además los diferentes métodos de comunicación, tanto punto a punto como colectivos, así como las posibilidades de modos que ofrecen, nos pueden aportar diferentes tiempos extra de overheads.

En las comunicaciones punto a punto pueden generarse diferentes overheads dependiendo de los métodos utilizados, uso de bloqueo o no bloqueo, y uso de técnicas de buffering.

En las comunicaciones sin buffer bloqueantes podemos encontrarnos con varios escenarios diferentes (ver figura 2.2), según los momentos de aparición de las primitivas de envío y recepción (dentro de los procesos que envía, sender, y el que recibe, receiver en la figura). En particular para diferencias de aparición de una respecto de la otra, se nos pueden producir tiempos de espera importantes, que están manteniendo en espera bien al emisor, o al receptor. Con lo cual este protocolo, en las comunicaciones, es más adecuado para situaciones donde se podría espera que las primitivas aparecieran simultáneamente. Pero si el entorno es de tipo asíncrono, es difícil prever que puedan darse estas situaciones, y en este caso será frecuente la aparición de tiempos de espera en emisor o receptor que provocaran a posteriori ineficiencias en la aplicación, si su frecuencia de aparición se mantiene a lo largo de la ejecución, o si se dan múltiples instancias de este problema.

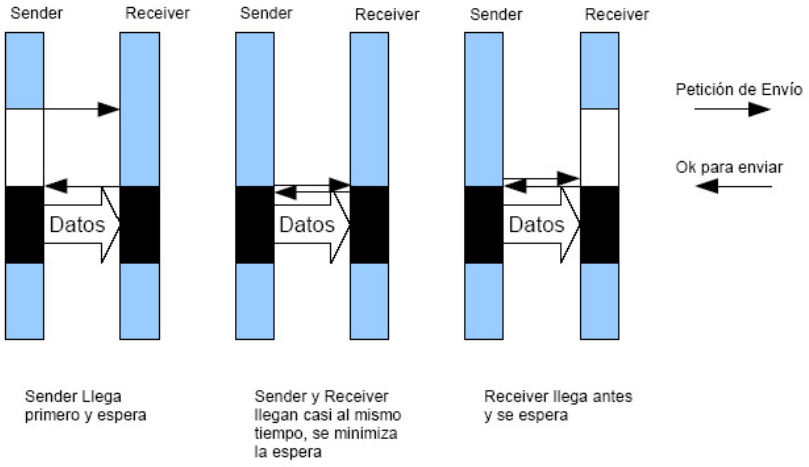

Figura 2.2: Escenarios en la operación envío/recepción bloqueante sin buffer

En las comunicaciones bloqueantes con buffer, por contra, tenemos una disminución importante de estos tiempos de espera, penalizada con tiempos por gestión de buffers (que se pueden disminuir si disponemos de hardware adecuado). Si la aplicación tuviera un alto nivel de sincronismo, seguramente serian más eficaces los bloqueos sin buffer (disminuimos esperas, penalizamos gestión de buffers), pero en las aplicaciones en general suele ser mejor la utilización de bloqueos por buffer, a no ser que las comunicaciones excedan las capacidades de los buffer utilizados.

En las comunicaciones no bloqueantes, la reducción de tiempos de espera puede ser importante, pero dependerá bastante que no hayan fuertes dependencias entre los datos usados como elemento de comunicación, y su proceso o cómputo posterior. Así como las estrategias de test usadas, ya sea por bucle de test, o bien por espera activa a posteriori. Se deberá valorar la granularidad del cómputo solapado posible en frente de las operaciones de comprobación necesarias. La existencia de hardware de comunicaciones en el sistema, puede mejorar este aspecto al eliminar el tratamiento de buffers, y permitir un solapamiento todavía mayor de cómputo.

Algunos nuevos trabajos como los de [Klu05], intentan definir valores limites (*thresholds*) para la ejecución de algunas primitivas de paso de mensajes, mediante experimentación con una serie de benchmarks. La idea es definir un comportamiento normal, en tiempo, de la primitiva, y en el momento que este fuera de estos limites de una distribución estadística dada (observada en los benchmarks), utilizarla como síntoma para iniciar la detección como origen de un problema de prestaciones. Esta aproximación es interesante, pero aduce de ser necesaria una caracterización del sistema, y del comportamiento de los benchmarks para cada plataforma, de forma parecida a lo que ocurría con otras aproximaciones como [Vet00].

Para un análisis más o menos detallado de los tiempos envueltos en un entorno de paso de mensajes, tendremos en cuenta las consideraciones anteriores, así como la existencia de los diversos componentes del sistemas (tareas, red, mensajes, canales, etc.) para definir un modelo de tiempos. Examinaremos una modelación habitual del tiempo [Dil95, Yan96, Rau97, Rau00] en paso por mensajes, teniendo en cuenta las primitivas explicitas habituales que se usan en las aplicaciones por paso de mensajes, y ampliar las consideraciones temporales realizadas en las secciones previas.

En el caso de una aplicación por paso de mensajes podemos expresar el tiempo, descompuesto en las actividades principales, generalmente como:

$$Tiempo\,total \;=\; T_{comp} \;+\; T_{comm} \;+\; T_{sys} \;+\; T_{io}$$

siendo

Tcomp el tiempo de computación. Tcomm el tiempo de comunicación. Tsys el tiempo de sistema Tio el tiempo de E/S.

En el caso de un sistema de paso de mensajes, teniendo en cuenta las primitivas existentes, podemos dividir el tiempo de comunicación (Tcomm) entre el tiempo dedicado a las primitivas de comunicación punto a punto (Tp2p) y el dedicado a operaciones colectivas y sincronización (Tcol), obteniendo así

$$T_{comm} \;=\; T_{p2p} \;+\; T_{col}$$

A su vez en el caso de las comunicaciones punto a punto, tendremos que tener en cuenta, la separación entre tiempo dedicado a envío, y el dedicado a recepción, se expresa así el tiempo como:

$$T_{p2p} = T_{send} + T_{recv}$$

Al analizar con más detalle los diferentes aspectos de la comunicación punto a punto, por ejemplo en el caso de la operación de recepción, la tendremos que desglosar teniendo en cuenta los diferentes fases envueltas en completar la operación:

- Tiempo idle (Tidle) debido a la posible espera (en caso bloqueante), hasta que se presente la operación correspondiente de envío.
- Tiempo transmisión (Ttrans), tiempo de recepción del mensaje.
- Tiempo de overhead (Tover), costes fijos asociados a la transmisión.
- Tiempo de contención (Tcont), tiempos asociados a posibles problemas de contención en la red de interconexión usada.
- Tiempos de copia en buffer (Tbuf), posibles tiempos asociados en la libreria, por el uso de técnicas de buffer, por la realización de copias de los mensajes recibidos entre los buffers de sistema, y los buffers de usuario.

$$T_{recv} = T_{idle} + T_{trans} + T_{over} + T_{cont} + T_{copy}$$

En el caso de recepción, podemos observar como afectaran diferentes posibles optimizaciones que podemos realizar sobre el tiempo. En el caso de múltiples operaciones de recepción, podemos agrupar los mensajes (agregación) con el objetivo de reducir el tiempo de transmisión [Alb99]. La posibilidad de uso de recepción asíncrona (no bloqueante), nos permite evitar los tiempos idle, al aprovechar el posible solapamiento entre comunicación y cómputo. Tover es normalmente constante, asociado al coste de la transmisión. En Tcont, el problema está en la compartición de los recursos de interconexión (ya sea entre diferentes aplicaciones y/o sistemas, o por las propias tareas de la aplicación).

Otra aproximación a los tiempos en el caso a punto a punto, es el análisis desde la perspectiva de las dos tareas participantes. En particular en este tenemos mecanismos directos de medición de los datos teóricos, mediante, por ejemplo, los benchmarks basados en envíos de mensajes mediante pingpong entre las dos tareas participantes. Para este análisis podemos observar el tiempo TAB de un envió de un mensaje de tamaño N de la tarea X a la tarea Y que lo recibirá.

$$T_{XY}(N) = T_{start} + T_{dsend}(N) + T_{net}(N) + T_{drecv}(N)$$

donde:

- Tstart : tiempo de arranque de la operación de envió, consumido en trabajo interno, gestión de memoria para buffers, establecer conexiones con el receptor. Se supone constante, ya que normalmente no depende del tamaño del mensaje.

- Tdsend : tiempo de comunicación de envió, en estar el mensaje disponible para envió en la interfaz de red.
- Tnet : tiempo del movimiento del mensaje desde la interfaz del emisor a la interfaz del receptor. Se considera solo desde la perspectiva de nivel usuario, como transmisión del mensaje, dependiendo de la implementación de las comunicaciones a bajo nivel, también podría incluir los acuse de recibo, y particiones de mensaje, u otros factores, dependiendo del modo de envío utilizado y su posible implementación.
- Tdrecv : tiempo necesario por el receptor para obtener el mensaje desde la interfaz de red.

Los tiempos Tdsend y Tdrecv, indican tiempo consumido en las funciones de libreria, hasta o desde disponer el mensaje en el interfaz de red. En el caso del emisor Tdsend puede expresarse como:

$$T_{dsend}(N) \; = \; T_{pack}(N) \; + \; T_{send}(N) \; + \; T_{syssend}$$

donde se tiene en cuenta los procesos de preparación del mensaje, y su disposición final en el interfaz de red. Aparece el tiempo de empaquetar el mensaje (codificarlo si tenemos soporte heterogéneo, copiarlo y la gestión de los buffers), proceder al envió del mensaje hasta dejarlo en el interfaz de red, y el coste adicional de llamadas a sistema dentro del código de la primitiva. Tanto el empaquetamiento como el envió pueden considerarse proporcionales al tamaño del mensaje, ya que su coste es lineal con el. Mientras el tiempo de llamadas adicionales puede considerarse constante. Se ha observado [Dil95] que dependiendo del tamaño del mensaje (N) el coste supera a la componente fija de sistema (en particular, la agregación de mensajes favorecería este punto [Alb99]). En este caso para ciertos tamaños del mensajes:

$$T_{dsend}(N) \; = \; N * (T_{pack} \; + \; T_{send})$$

Experimentalmente (en tests de tipo pingpong) podemos medir globalmente en el sistema el envío de un mensaje de una tarea a otra, y el retorno en las mismas condiciones, indicándolo como TXYX (envío de la tarea X a la Y y retorno a la misma X). Se observa así el tiempo de comunicación punto a punto como:

$$T_{XY}(N) \; = \; \frac{T_{XYX}}{2}$$

2.5. Aproximación clásica: Visualización

La aproximación clásica del análisis de prestaciones, se ha basado en herramientas de visualización. Primero, los programadores desarrollan y depuran sus aplicaciones. Luego, las ejecutan con la ayuda de herramientas de monitorización

que recolectan la información sobre el comportamiento de la aplicación. Luego, normalmente en una fase posterior a la ejecución, la herramienta de visualización muestra la información recolectada utilizando diferentes vistas (como diagramas de Gantt, diagramas de barras, de tarta, etc.). Hay múltiples herramientas de visualización, que ofrecen diferentes aproximaciones de interficie de usuario y tipos de vistas mostradas, permitiendo al usuario navegar intuitivamente entre diferentes vistas de los datos de su aplicación. Con estas herramientas, podemos obtener una aproximación rápida al rendimiento de la aplicación, y a sus principales problemas. En esta, y próximas secciones discutiremos sobre la aproximación de análisis basada en visualización, y las técnicas empleadas para la recolección de datos y su visualización.

Si los programas paralelos, no aprovechan correctamente los recursos disponibles del sistema, necesitamos la monitorización para descubrir los factores por los cuáles tienen perdidas de rendimiento. A las aplicaciones se les añade cierta información de monitorización o bien de instrumentación, que dependiendo de su tipo puede seguir diferentes técnicas:

- De tiempo: se utiliza el tiempo de ejecución para detectar donde el programa paralelo gasta la mayor parte del tiempo. Normalmente no se incluye ningún tipo de explicación.

- Contadores: se utiliza para contar el número de veces que suceden determinados eventos.

- Muestreo (sampling): a través de esta técnica, se obtienen ciertas medidas periódicas del estado de la aplicación. Normalmente se detiene la aplicación a intervalos regulares para tomar las medidas oportunas.

- Trazas: se obtienen secuencias de información asociadas a ciertos eventos que se dan en las aplicaciones paralelas.

A partir de las medidas tomadas pueden obtenerse información a modo de resumen, o en forma de visualización gráfica, de las prestaciones obtenidas por la aplicación paralela en un sistema de cómputo paralelo.

Pueden utilizarse los índices de prestaciones (vistos en el primer capítulo) o crear unos índices virtuales nuevos, que se adecuen a la aplicación concreta. Por ejemplo, triángulos renderizados por segundo en un sistema gráfico, frames en un sistema de animación, en este caso por ejemplo estaríamos tomando medidas de ratio de ejecución orientadas a la aplicación concreta.

En la aproximación clásica de visualización de estos datos, pasamos (como veremos) por unas fases de: a) recolección de datos, b) transformación de datos, y c) visualización de los datos. Las cuáles se suelen insertar dentro de un proceso cíclico por parte del desarrollador para intentar conseguir las prestaciones más adecuadas de la aplicación.

Normalmente las herramientas visuales [Sum00], usualmente, para intentar explicar que sucede en la aplicación ejecutada, presentan su información bien en forma textual, o bien de forma gráfica, generalmente usando diagramas 2D, o también en 3D en algunas herramientas que utilizan conceptos de realidad virtual.

Con la herramienta de visualización se espera poder identificar los problemas de la aplicación con las prestaciones de las tareas y sus comunicaciones, la estructura de los algoritmos utilizados y su implementación. En general una herramienta de visualización tendría como valor permitir al desarrollador comparar fácilmente el patrón de ejecución observado con lo que el desarrollador espera del análisis y diseño de la aplicación.

Para visualizar la información de las métricas de prestaciones (e información asociada o post procesada), se utilizan diagramas que entran dentro de las categorías siguientes: Grafos de comunicaciones, resúmenes estadísticos, diagramas de topologías, resúmenes estadísticos de comunicaciones, y árboles de llamadas.

Entre las diferentes categorías, podemos observar:

Grafos de comunicación Estos diagramas son los más utilizados, y típicamente en la forma de un diagrama de Gantt, en el cual se visualiza el estado de la ejecución, con ejes de representación temporal y tareas participantes, junto con las comunicaciones representadas como lineas entre las tareas. El estado de cada tarea, a lo largo de su ejecución, es habitualmente expresado con colores. Se proporciona de esta manera relaciones entre el estado de las tareas y sus comunicaciones. Algunos problemas, es que no escalan correctamente respecto a gran número de procesadores (y tareas) [Cou93, Sum00], o la eficiencia de mostrar volúmenes de comunicación en periodos de tiempo cortos.

Resúmenes estadísticos por tarea Normalmente realizados por diagramas de tarta, que intentan proporcionar porcentajes de tiempo sobre cada tarea en sus diferentes estados o comunicaciones. Normalmente nos permite establecer comparativas rápidas entre las diferentes tareas, para detectar comportamientos anómalos (o especializados según el diseño de la aplicación) respecto a la media.

Diagramas de topologías Éstas son normalmente representadas mediante nodos de tareas que son conectados a partir de una vista en anillo o malla. Visualizándose normalmente en el formato de una animación, donde el estado de la tarea colorea el nodo, y los enlaces que van surgiendo según se realizan las comunicaciones.

Resúmenes estadísticos por comunicación Típicamente visualizadas en forma de matrices de N x N tareas, donde se representa por celda, la comunicación entre las dos tareas a que se hace referencia, normalmente en bytes o Kbytes intercambiados, según la dirección de comunicación de una tarea de un eje a la del otro eje. Se pueden usar colores para las celdas según los tamaños, también para facilitar la visión de comunicaciones de gran volumen. Debido a su

naturaleza acumulativa es difícil detectar problemas transitorios, o cíclicos durante la ejecución. Aunque podría mejorarse significativamente mediante análisis por fases, o por medio de métodos de detección de patrones. Por otra parte, otras visualizaciones complementarias de las comunicaciones, son los resúmenes según tamaños de paquetes empleados, y el número de estos usados. O bien ratios de información de E/S por tarea. Otra aproximación útil en el paso por mensajes, es la visualización de las situación de las colas de mensajes a lo largo de la ejecución en cada usuario.

Arboles de llamadas Se trata de un diagrama en árbol que muestra las llamadas realizadas en la aplicación sobre sus diferentes elementos, ya sean objetos, subrutinas, o incluso a más bajo nivel al llegar a bucles y condicionales. Pudiéndose utilizar para depurar el algoritmo, viendo si su ejecución refleja lo que esperaba el desarrollador.

Como veremos en los siguientes capítulos, y como ya avanzamos al inicio, las herramientas visuales no son suficientes para afrontar la gran complejidad de las aplicaciones paralelas actuales, así como aportar al desarrollador el conocimiento necesario, para llevar a cabo de forma manual (solo ayudado por estas herramientas visuales) la optimización de prestaciones de las aplicaciones. Además hay que superar el ratio de conocimiento y tiempo aplicado en las optimizaciones respecto de las prestaciones conseguidas finales. Se hacen necesarias herramientas que den pasos más alla y que permitan asistir y automatizar el proceso de forma que se mejoren los aspectos comentados.

En los capítulos siguientes, analizaremos las diferentes fases del análisis de prestaciones para paradigmas por paso de mensajes, así como los conceptos que deben implantar las herramientas que ofrezcan análisis automático de prestaciones.

2.6. Técnicas de Monitorización: Tracing vs Profiling

Los datos de prestaciones que se necesitan obtener para el análisis posterior son de diferente naturaleza, tanto cualitativa como cuantitativa. Ya sea mediante datos aportados: a) Comprobar la existencia de unos sucesos y su orden temporal; b) Su caracterización respecto al número de ocurrencias y parámetros; c) Respecto a su localización espacial (en el código); d) Respecto a diferentes entidades de la aplicación estáticas (regiones de código) o dinámicas (call paths).

Una distinción básica en las herramientas de análisis de prestaciones es la orientación que toman en las fases de recogida de esta información, mediante técnicas de muestreo (basadas en sampling periódico o en uso de contadores especializados) o bien en la observación de sucesos, eventos que se han producido, y en recoger la información asociada a cada uno de ellos (los parámetros de cada evento).

Al primer conjunto se les suele denominar como herramientas basadas en profiling, aunque también se habla de muestreo, contadores o estados. Básicamente

se trata de recoger datos, para su visualización o análisis on-line durante la ejecución de la aplicación (aunque algunas herramientas también los usan a posteriori de la ejecución). Estos datos intentan ser lo más pequeños posibles, o se enfocan hacia aquellas partes o puntos de la aplicación que nos interesan especialmente, para minimizar al máximo el tiempo adicional causado (la perturbación, o ruido introducido) en la ejecución de la aplicación por los sistemas de medida.

Al segundo conjunto se las denomina como herramientas basadas en traza. Normalmente en estas se utiliza una aproximación de tipo post-mortem, ya que en estos casos los datos son recogidos (habitualmente) durante la ejecución de la aplicación, pero las herramientas, para evitar perturbación, los utilizan para la visualización o análisis, con posterioridad a la ejecución de la aplicación. Normalmente las técnicas basadas en traza tienen una perturbación grande en la aplicación. Dependiendo en gran medida de los parámetros que se asocien a cada evento, y si estos son de obtención directa o se necesitan algunos cálculos o procesos extra para obtenerlos. Con los sistemas de traza se intenta maximizar la información obtenida, teniendo en cuenta que el proceso importante de análisis se realizará a posterioridad en una fase postmortem.

En el caso de profiling hemos de tener en cuenta que lo que acabamos obtenido es información de prestaciones en forma de sumarios, que en la mayor parte de los casos pueden ser suficientes para detectar muchos de los problemas de prestaciones frecuentes que ocurran durante la ejecución de la aplicación. A pesar de esto, hay problemas de prestaciones que no son visibles a este tipo de recolección de información, por su desaparición en las vistas resumen de datos.

En contraste, las trazas permiten la reconstrucción del comportamiento dinámico en términos de los eventos, y producen una visión mucho más detallada de lo que ha sucedido durante la ejecución de la aplicación. Por otra parte las técnicas de profiling pueden tener la ventaja de escalar mejor según el sistema crezca (en procesadores) ya que básicamente proporciona resúmenes de información estadística. En resumen, los métodos de profiling, tienen la ventaja de provocar poca perturbación en la ejecución de la aplicación, y pueden relacionarse fácilmente con el código (respecto las regiones donde se ha recolectado la información) pero muchas veces es difícil saber que sucede realmente a partir solo de esta información.

2.7. Visualización basada en Tracing

Como hemos visto, en el caso de las herramientas basadas en tracing, lo que pretendemos es generar una secuencia de registro de eventos ocurridos en la aplicación en forma de uno (o varios) ficheros de traza. En ellos hacemos constar todos los eventos recogidos del sistema, junto con la información de los parámetros de cada uno de ellos, referentes a su acción y sus atributos.

Las trazas son colecciones de registros de los eventos sucedidos durante la ejecución de la aplicación. Normalmente estos eventos se acompañan de información, tal como el punto preciso de tiempo en que ha sucedido, la localización en el sis-

tema hardware-software (nodo, procesador, tarea, thread si fuera el caso), el tipo de evento, así como otra información adicional variable necesaria según el tipo de evento.

En el caso que nos ocupa, de sistemas basados en paso de mensajes, normalmente se capturan eventos relacionados con el uso de las primitivas, y que son guardados en el momento de su ocurrencia, cuando se producen las llamadas. Por esta razón, se necesita instrumentar las primitivas para poder capturar su tipo, y los parámetros que afectan a su ejecución.

Normalmente toda esta información es guardada en los ficheros de traza, que suelen tener un tamaño bastante grande (dependiendo de las interacciones de la aplicación y de su tiempo total de ejecución). En concreto pueden aparecer problemas [Fre02] de capacidad de almacenamiento de las trazas, ya sea de forma individual o bien trazas de una misma aplicación bajo ejecuciones con condiciones cambiantes de los datos de entrada. Las herramientas de visualización pueden presentar problemas en el manejo de trazas grandes (debidas al número de eventos, al número de procesadores, o tareas muy grandes en ejecución), que provocan tiempos de respuesta grandes de cara al usuario. A estos problemas se les suele denominar como problemas asociados a la escalabilidad de los sistemas de traza. Algunas propuestas para minimizar estos problemas se basan en eliminar información redundante, como por ejemplo iteraciones iguales dentro del fichero, o bien a formatos con información comprimida, o bien tratamientos iterativos por porciones de traza [Esp00], o otras soluciones basadas en crear estados o información estadística de resumen a partir de la traza, o que combinan la información del grafo de llamadas [Knu05b].

Habitualmente se utilizan diversas técnicas para minimizar la perturbación del sistema, introducida por el sistema de generación de traza en la ejecución de la aplicación, como por ejemplo, la inclusión de buffers en memoria, que se vuelcan en ciertos intervalos o solo al final de la aplicación [Mai95]. También es común introducir fases de sincronización de tiempos global, antes de la ejecución, y fases de post-tratamiento de los ficheros de traza al finalizar la aplicación. Otras técnicas de reducción de perturbación, trabajan con una traza selectiva, se filtran solo algunos eventos del total de los posibles. O también se distribuye este proceso en una arquitectura jerárquica [Als96, Buy00] dentro del sistema monitorizado.

Una de las principales ventajas de los ficheros de traza, es la alta cantidad de información que se puede obtener de la ejecución de la aplicación, en particular de la localización tanto temporal como espacial de los diferentes eventos capturados. Se hace así posible una reconstrucción, o incluso reproducción, completa de la aplicación y su comportamiento en tiempo de ejecución. Además es fácil, si así se desea, obtener información de tipo resumen, parecida a las técnicas de profiling, ya que tenemos un detalle mayor de la información, y podemos crear índices o resúmenes, mediante tratamientos estadísticos de la información obtenida por los ficheros de traza.

Comentamos también brevemente algunas herramientas de análisis de prestaciones por visualización basadas en traza:

2.7.1. Jumpshot

Se trata de una herramienta [Zak99] disponible para el entorno MPICH, que ha pasado por diferentes versiones, upshot, nupshot, y las actuales revisiones Jumpshot-X (X=1,2,3,4). Básicamente captura los eventos que se produzcan en aplicaciones MPI, de cara a visualizar un diagrama de Gantt que representa la ejecución de las tareas en diagramas de linea de tiempo, y las interacciones entre tareas que participan en comunicaciones de tipo punto a punto o colectivas, también es capaz de capturar eventos que defina el usuario, aparte de las propias llamadas. La visualización se ha ido enriqueciendo con las diferentes versiones, desde los diagramas de Gantt, a varios sumarios de tiempos por tarea o procesador de los diferentes eventos y tiempos, así como agrupar o desagrupar el efecto de diferentes llamadas que forman la ejecución. En particular es importante los desarrollos realizados en los formatos de traza, desde formatos ASCII básicos de eventos (ALOG), a diferentes formatos comprimidos (CLOG), y nuevos formatos basados en resúmenes o estados mediante sampling (SLOG1), o basados en entidades para su visualización (SLOG2). Cada una de las revisiones del visualizador Jumpshot están relacionados con la visualización de uno de estos formatos.

2.7.2. Paraver

Es una herramienta diseñada en la universidad Politécnica de Cataluña (UPC), se trata de un visualizador de trazas, que tiene un amplio rango de sistemas de programación distribuidos (MPI, OpenMP, híbrido de ambos, contadores hardware, Java, actividad del operativo) sobre varias plataformas UNIX. Visualiza trazas en lineas de tiempo, de eventos producidos en llamadas del sistema (o definidas por el usuario), así como también alguna información de contadores hardware, o funciones llamadas. En diferentes diagramas de tipo acumulativo. Una particularidad del sistema es la capacidad de definir métricas a visualizar por parte del usuario, mediante combinaciones u operaciones con las métricas obtenidas por la herramienta. Esta herramienta también es usada como visualizador de otra herramienta denominada Dimemas, que es usada para análisis de prestaciones mediante simulación de la aplicación paralela.

2.7.3. Paragraph

La herramienta de monitorización Paragraph, desarrollada en la Universidad de Illinois, permite recoger información gráfica de funciones de librería utilizadas. Produciendo una dinámica, detallada y gráfica animación del comportamiento de la aplicación paralela monitorizada. Reproduciendo gráficamente los eventos que hayan sucedido durante la ejecución del programa.

La información que muestra esta herramienta está dividida en tres aspectos: 1) Utilización del procesador; 2) Comunicación entre las tareas que forman parte de la aplicación paralela; 3) Información sobre las propias tareas.

Paragraph utiliza para recoger la información, la librería MPICL (basada en PICL) que permite recolectar la información de los eventos de comunicación y los eventos que defina el usuario en aplicaciones paralelas basadas en paradigmas de paso de mensajes. MPICL instrumenta el código del programa por medio de rutinas C, que utilizan llamadas de temporización (respecto al reloj del sistema), y guardan información obtenida de los eventos (número de ocurrencias, tiempo de comunicación, y eventos de usuario para cada procesador). La información es guardada en buffers internos, que solo se vuelcan periódicamente a disco, para conseguir una intromisión menor en el tiempo de la aplicación, .

2.7.4. Xpvm

Es una de las herramientas de visualización más utilizadas para PVM, ya que XPVM [Gei94, Koh96] actualmente está integrada con la instalación de PVM (o puede añadirse fácilmente en otras versiones anteriores). Integra las herramientas de visualización basadas en traza, junto con un entorno que permite interactuar, de forma similar a la consola PVM, para lanzar aplicaciones, y manejar la maquina virtual definida, permitiendo interactivamente la creación, o eliminación, de tareas y máquinas.

En este caso la herramienta de monitorización, permite mediante configuración de los sistemas de traza internos a la librería PVM, recolectar los eventos de las primitivas PVM y redireccionar la información a la salida gráfica de la herramienta en tiempo real (aunque también puede realizarse postmortem a la ejecución). Esta herramienta es usada para mostrar en tiempo real (o postmortem), la comunicación de mensajes entre tareas (pueden visualizarse los tamaños y estado de las colas de mensajes) y las operaciones realizadas entre grupos de tareas. En la figura 2.3 se muestra la ventana principal de Xpvm, con información sobre la configuración de máquinas y el tiempo de los eventos de comunicación entre tareas.

2.7.5. ICT

Vampir [Nag96], recientemente renombrada a Intel Cluster Toolkit (ICT), es un desarrollo comercial de la empresa Pallas (después comprado por Intel). VAMPIR (Visualization and Analysis of MPI programs), está orientado a la visualización de aplicaciones paralelas MPI. Evalúa, mediante técnicas sencillas de contadores de tiempos y de categorías de primitivas, el rendimiento de subrutinas o bloques de código de una aplicación MPI, e identifica donde se produce el cuello de botella en las comunicaciones. Esta herramienta, mediante múltiples vistas gráficas, muestra información sobre las comunicaciones, parámetros y el rendimiento de la aplicación. Un ejemplo de esta información se muestra en la siguiente figura (2.4).

Vampir también utiliza ficheros de traza para visualizar posteriormente el comportamiento de una aplicación después de su ejecución. Esta herramienta utiliza una librería denominada VAMPIRtrace, que se encarga de las operaciones de monitorización mediante traza de la aplicación MPI, se pueden monitorizan las comu-

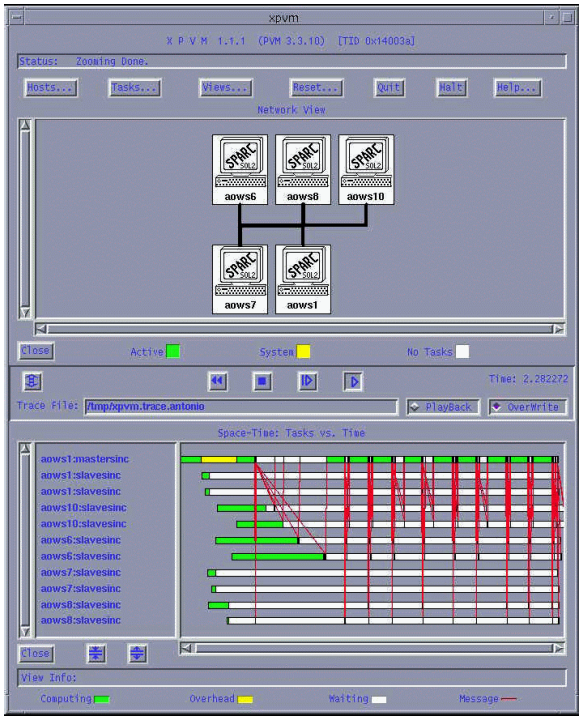

Figura 2.3: Estado de una aplicación PVM en la herramienta XPVM

nicaciones punto a punto, colectivas, entrada y salida, y las funciones que haya definido el usuario, definiendo eventos de entrada y salida a las regiones donde se producen cada uno de los sucesos. También se utilizan diferentes contadores de tiempos, y de tamaño de mensajes para realizar cierto profiling que luego se muestra en forma de vistas gráficas como las observadas en la figura anterior, referidas a estadísticas de comunicaciones (dependiendo de tiempos y longitud de mensajes), y métricas de ejecución de rutinas. Todas estas métricas pueden consultarse de forma global (para toda la aplicación) o bien para cada proceso de esta. Además es posible comparar diferentes ejecuciones de la misma aplicación, para determinar las variaciones de las métricas.

Vampir visualiza las trazas de eventos de los programas de paso de mensajes situándolos en lineas de tiempo asociadas a cada proceso, se indica cada uno los estados de ejecución por medio de un color. Utiliza flechas para indicar las conexiones entre procesos, para los pasos de mensajes de punto a punto, o lineas que conectan los procesos que intervienen en una operación colectiva (el tipo es indicado también por un color) de grupo. Tiene capacidad de zoom del diagrama, para visualizar la información en diferentes incrementos de tiempo, y desplazamientos a lo largo de la ejecución de la aplicación.

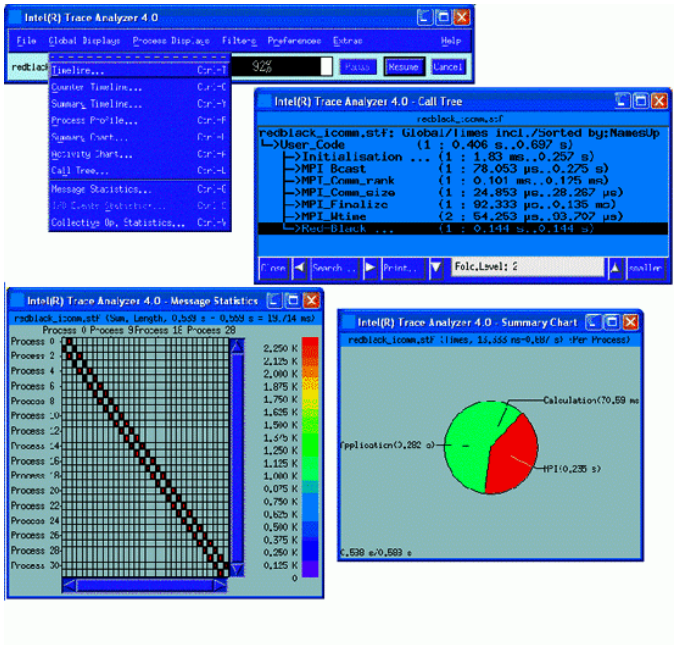

Figura 2.4: ICT: Algunos Diagramas disponibles

2.8. Visualización basada en Profiling

Las herramientas orientadas a profiling suelen ser de naturaleza dinámica, en el sentido que suelen captar sus datos, y mostrarlos, durante la ejecución de la aplicación paralela o a posteriori. El objetivo es tomar medidas, poco perturbadoras, de algunos índices predefinidos en un modelo de prestaciones, que nos sirvan para obtener una vista rápida de como está funcionando la aplicación, y poder así deducir cual es la localización o causas de su falta de rendimiento.

Se utilizan métodos de sampling, contadores, o resumen de estados, para tomar una serie de medidas que pueden mostrarse en el momento de la ejecución, o también analizarse (en algunos casos) a posteriori en postmortem.

La idea básica de los profilers es mapear los datos obtenidos respecto de las entidades estáticas o dinámicas de la aplicación. Como por ejemplo recoger medidas diversas de cada función de la aplicación, y mapear los índices más bajos o altos sobre las funciones donde ocurran. Los objetivos básicos son poder determinar donde se consume el tiempo respecto los diferentes módulos de la aplicación, y encontrar las secciones de código más frecuentes o con el mayor tiempo consumido.

En estas medidas, normalmente se utilizan medidas de tiempos durante la ejecución. Estas medidas son realizadas por llamadas especificas para medida de tiempos en librerías estándar o bien especificas de un entorno. Llamadas como *clock(), times(), gettimeofday(), gethrtime()* o *getrusage()* (en lenguaje C), o para entornos concretos como *MPI_Wtime()* (en MPI), o en Fortran90, *qw_time(), system_clock()*.

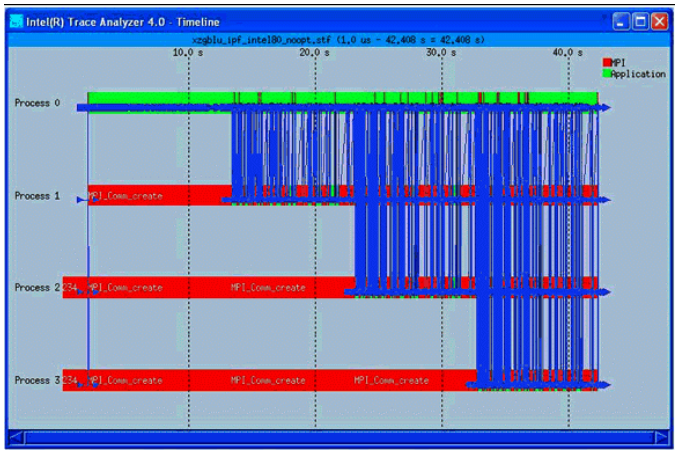

Figura 2.5: ICT: Lineas de tiempos por proceso

Estas llamadas nos permiten, mediante inserción en el código, obtener tiempos que representan la ejecución de funciones, bucles, o bloques específicos de código, o bien la aplicación entera.

Para las técnicas basadas en contadores, muchas veces son usados los llamados contadores hardware, que proporcionan estadísticas de las operaciones realizadas por las CPUs. Generalmente, en las CPUs actuales, se dispone de contadores integrados en forma de registros internos, que acumulan la aparición de determinados eventos hardware, como señales especificas, funciones del procesador, o relación con las memorias cache (fallos o aciertos en los accesos) o el sistema de memoria virtual, y la paginación y/o segmentación. En particular, la monitorización de estos contadores nos pueden ofrecer índices de la eficiencia de la traslación del código en el arquitectura final que dispongamos. En particular existen varias librerías para acceder a estos contadores, como PAPI y PCL.

Las dos técnicas básicas del profiling son el muestreo (sampling), y la instrumentación. En la primera se intenta una aproximación estadística en la cual se observan (o toman) periódicamente una serie de medidas sobre elementos predefinidos de la aplicación. Bien sea de forma acumulativa, como en forma de distribución. En el muestreo se pueden presentar problemas debido al concepto de cobertura del código de la aplicación, ya que una ejecución puede no cubrir gran parte del código de la aplicación, y dejar así sin medidas una parte de la aplicación, que podría ser la causante de problemas a posteriori en otras ejecuciones.

Respecto a la instrumentación, normalmente se inserta código directamente en la aplicación en ciertos puntos, para obtener puntos de medida concretos o bien seleccionar partes de la aplicación que se consideran importantes de monitorizar. Pueden establecerse puntos genéricos de instrumentación, como entradas y salidas de funciones, o dejar al usuario que introduzca regiones de especial interés.

Otra idea habitual es el profiling basado en estados, la idea es que utilizando

cualquiera de las dos técnicas anteriores, crear una serie de estados resumen de forma periódica en el tiempo, o en localización espacial (regiones de código), de manera que se tenga un resumen de ciertos parámetros (o índices) medidos, de manera que sea más o menos fácil su comparación, y el estudio estadístico de la distribución de los valores de los estados a lo largo de la ejecución de la aplicación.

También a veces se hace servir la idea de evento en los sistemas de profiling, como la toma de un estado concreto, o valor de algún índice, o como la variación, aparición o desaparición de la medida. En este sentido también el profiling podría definirse como una sucesión de eventos, con la localización espacial y temporal de estos.

En definitiva, las herramientas de profiling nos permiten tomar medidas en bruto de la aplicación, presentándonos (dependiendo de modelos de estado, o índices concretos) un borrador de lo que ha sucedido en la aplicación a lo largo del tiempo, y por tanto de las prestaciones que estamos obteniendo. Los sistemas de profiling tienden a introducir una perturbación limitada en la ejecución, y requerir poco almacenamiento. Además la perturbación puede disminuir si limitamos a priori las regiones temporales y espaciales de análisis a las que sean de interés, o bien si se realiza este proceso de localización de forma dinámica y guiada según como se estén produciendo los índices en ejecución. Básicamente se tiene que establecer un compromiso entre la demanda de información, y la perturbación que la monitorización este causando.

En las secciones posteriores comentaremos algunas herramientas basadas en profiling con más extensión. Incluimos dos herramientas, más brevemente, como explicación de algunos de los conceptos básicos comentados:

Xprofiler (ver figura 2.6) es un frontend gráfico a la herramienta de profiling GNU gprof, que nos permite recoger información de tiempos de las funciones de nuestra aplicación, así como el grafo de llamadas (call graph) que se genera entre ellas. Ofrece soporte también para aplicaciones paralelas, mediante la generación de un fichero de profile para cada proceso, que puede verse individualmente o de forma conjunta. Presenta un diagrama general de la aplicación, en forma de grafo de llamadas, donde nos podemos desplazar, y obtener una vista más detallada de lo que sucede en un subconjunto determinado de rutinas. Cada rutina es ofrecida en forma gráfica como un rectángulo, donde los tiempos inclusivos (de ella y sus subrutinas) o exclusivos (ella sola) forman la anchura y altura del rectángulo. Se conectan las rutinas que llaman (o son llamadas) mediante flechas indicando el sentido de la llamada.

Cray Apprentice [Cra94] es una herramienta basada en la recolección de información de profiling, para supercomputadores del fabricante Cray (X1, XT3 y otros). Ofrece estadísticas de llamadas de paso de mensajes y su distribución a lo largo del código, diagramas de linea de tiempos, para los procesos, coloreadas dependiendo del tipo de actividad. Recientemente, se ha presentado Cray Apprentice 2 (ver figura 2.7) que mejora sensiblemente los diagramas mostrados, incluyendo nuevos de visualización de call graphs, y añade a la herramienta clásica de visualización, técnicas de detección y análisis de problemas de prestaciones.

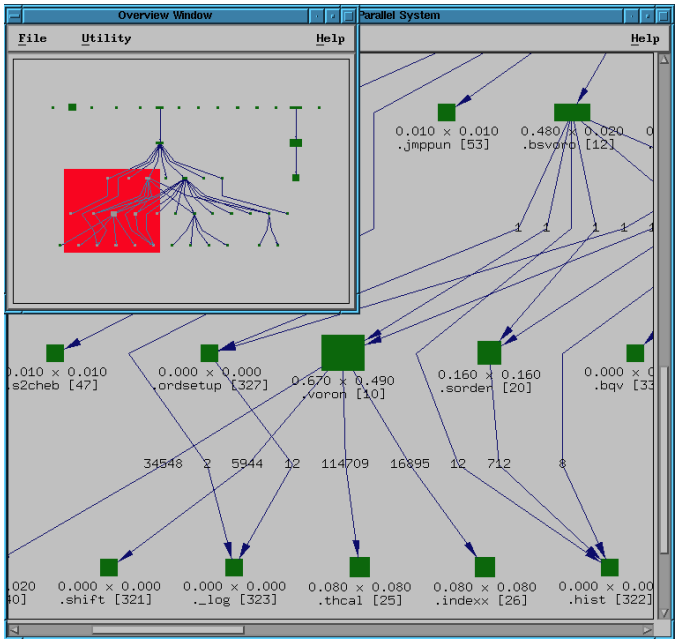

Figura 2.6: Xprofiler: diagrama general y detallado de los caminos de llamada

2.8.1. Tau

Es una herramienta [Moh94, Moo01] de profiling por medio de instrumentación en la cual se pueden tomar medidas de profiling tanto de contadores hardware como de parámetros en tiempo de ejecución basados en las medidas tomadas en rutinas, en bloques de código o a nivel de sentencia.

Se permite insertar anotaciones por parte del desarrollador en el código fuente de manera que indique aquellas zonas a las que se quiere instrumentar. También dispone de capacidades de reemplazan funciones de librería, por otras instrumentadas, o bien mediante un sistema basado en preprocesador instrumentar de forma automática código fuente de C, C++ y Fortran. Puede instrumentar las funciones de entornos MPI, mediante técnicas de substitución de funciones, y puede también proporcionar uso de instrumentación dinámica, mediante DyninstAPI [Buc00].

2.8.2. SvPablo

La primera versión de este entorno, denominada Pablo [Ree93], fue desarrollada en la Universidad de Illinois, incluyendo instrumentación, y representación gráfica 2D (junto con algunas técnicas más especiales de realidad virtual, y representación por sonido) de los datos recolectados. Proporcionaba instrumentación manual, y una interface gráfica para la especificación de los puntos de instrumentación. Permitía capturar llamadas a procedimientos, comunicaciones entre procesos,

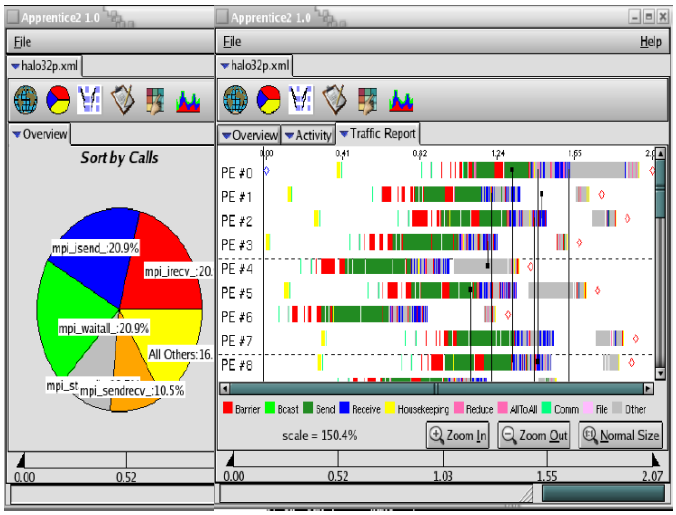

Figura 2.7: Cray Apprentice: Porcentajes de llamadas y diagrama de linea de tiempo

y operaciones de entrada/salida. Se presentaban varios gráficos de resumen de los datos recolectados, pero en particular tenia la posibilidad de generar histogramas de los datos capturados por diversos tratamientos estadísticos. Particularmente, disponía de la representación en realidad virtual de la aplicación, creando visualizaciones tridimensionales de los datos y comunicaciones de las diferentes tareas de la aplicación, así como uso de sonido para representar fases o comunicaciones entre estas. Todos estos análisis podían guardarse en un formato compacto denominado SDDF, que permitía crear estados de información, y compactarlos para minimizar la información incluida. La instrumentación incluida incluía medición de intervalos de tiempo, y contadores, normalmente definidos interactivamente por el usuario. También se incluían controles del nivel de instrumentación de cara a minimizar la pertubación. durante la ejecución.

El sistema, de la siguiente generación, SvPablo [Ros98] consiste en varios componentes (de los originales de Pablo), junto con una nueva interface gráfica, para instrumentar código fuente y observar datos de prestaciones. Fue diseñado para proporcionar captura de datos de prestaciones, análisis, y visualización de los datos para aplicaciones, escritas en diferentes lenguajes, y ejecutándose en una amplia variedad de entornos secuenciales y paralelos. Soporta instrumentación automática, en diferentes plataformas para códigos C y Fortran 77/90, y de High Performance Fortran (HPF con compilador PGI).

Para las aplicaciones de paso de mensajes, puede utilizarse MPI (en particular MPICH), y utiliza las librerías PAPI como interfase para obtener los contadores en tiempo de ejecución, y se pueden instrumentar las llamadas concretas de MPI a interceptar para el almacenamiento en el formato SDDF de Pablo.

En la figura 2.8 podemos observar algunos de los datos de contadores que puede mostrar sobre las regiones de código, y algunas métricas de una función, sobre tiempos, contadores y número de llamadas realizadas.

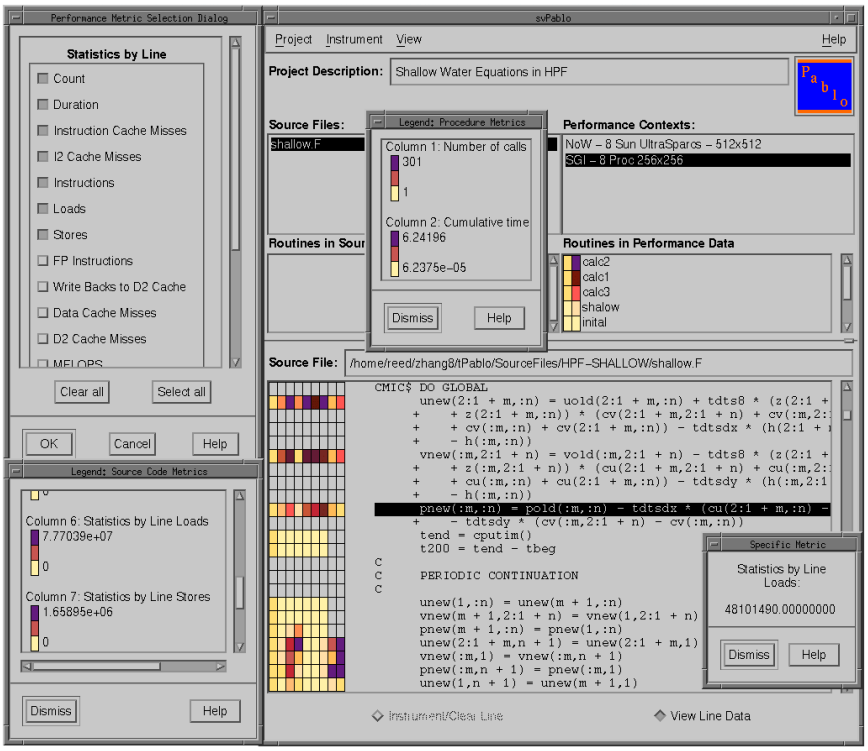

Figura 2.8: Interfaz Gráfica de SvPablo junto con relación de métricas

2.8.3. Paradyn

Paradyn es una herramienta que monitoriza el rendimiento de aplicaciones paralelas. Fue desarrollada en la universidad de Wisconsin y en Maryland. Proporciona monitorización y tiene componentes de análisis automático on-line (como veremos en el siguiente capítulo). Utiliza técnicas de profiling por instrumentación, ya que se emplea monitorización dinámica para instrumentar el código en ejecución de la aplicación.

Paradyn inserta y modifica instrumentación en tiempo de ejecución, por medio del uso de la librería DyninstAPI [Buc00, Hol02], sin tener que modificar el código fuente del programa, mediante instrumentación de la imagen de la tarea en tiempo de ejecución. Permite, entre otras técnicas, la inserción de pequeños trozos de código (*snippets*), a la entrada y salida de las funciones.

Paradyn proporciona análisis del rendimiento automático de las aplicación que se está ejecutando, intentando identificar las partes de esta que consumen más re-

cursos. Para ello considera un modelo de búsqueda de cuellos de botella denominado w3 (porque, donde y cuando) basado en realizar una serie de consultas referentes a:

- Porque la aplicación tiene mal rendimiento.
- Donde se produce el cuello de botella en el uso de recursos (CPU, ES, comunicaciones).
- Cuando ocurre el problema del cuello de botella.

El objetivo de esta herramienta es aislar de forma rápida y precisa el problema sin tener que examinar un elevado número de información.

Paradyn contiene un módulo llamado consultor del rendimiento que libera al usuario de tomar decisiones sobre el control de los datos de su aplicación. Este consultor busca cuáles son los problemas de la aplicación, decide cuáles son los datos que se tienen que almacenar, y cuando debe aplicar los cambios a la aplicación en su tiempo de ejecución. Mientras se realiza el proceso informa al usuario de los cambios realizados. En la figura podemos observar un ejemplo de salida que se muestra al usuario tras finalizar la fase de búsqueda.

La implementación de Paradyn, se divide en 3 partes: el controlador de Paradyn, los demonios y el proceso de aplicación. Cada uno de ellos:

- El controlador se encarga de realizar búsquedas de los cuellos de botella y soporta la interfaz de usuario para el resto del sistema.
- El demonio aisla las dependencias especificas de las máquinas y a la vez proporciona una etapa que relaciona el controlador con el proceso de aplicación.
- El proceso de aplicación está controlado por el demonio que se encarga de insertar de forma dinámica la instrumentación.

Paradyn además proporciona herramientas de visualización de la monitorización hacia el usuario. Esta herramienta denominada Performance Visualizations, informa del rendimiento del programa y del trabajo realizado por Paradyn en su tiempo de ejecución, visualizando los datos que se han estado obteniendo en cada instante de la ejecución del programa.

2.9. Herramientas de visualización vs automáticas

En las herramientas de monitorización, se insertan funciones o código de instrumentación en puntos determinados del programa paralelo, para recoger la información clave del funcionamiento. En la mayoría de los casos, esta información es depositada en un fichero de traza o de resumen de profiling, que la herramienta de visualización interpreta, para mostrar de forma gráfica, más entendible al usuario,

los campos más interesantes de la información recopilada. Ya sea en forma de visualización de las interacciones (cómputo, comunicaciones) de las tareas, o bien en forma de gráficas resumiendo algunos índices significativos de la aplicación.

La mayor parte de estas herramientas se basan en el proceso de recoger información, para posteriormente generar una serie de vistas gráficas diversas [Hea95, Sum00] de cara al usuario final. Básicamente se realizan tres fases, consistentes en la adquisición de datos, la transformación de estos, y la visualización, junto con métodos para manejar esta visualización como zooms y búsquedas, y habilidad en algunos casos de correlacionar varios diagramas, o mostrar desde más de una perspectiva los comportamientos observados.

El usuario final habitualmente tiene que realizar directamente el análisis del programa 2.9. Esto implica tener conocimientos avanzados de los sistema hardware y software implicados en la ejecución de la aplicación paralela. Tiene que pasar por un procedimiento de interpretación de la información obtenida por la herramienta, y utilizar una serie de heurísticas para comprender los índices que se le muestran en las gráficas visuales. Muchas veces la cantidad de información es enorme, teniendo múltiples vistas de varios índices. Estos índices suelen ser de bajo nivel, y muchas veces no directamente relacionados, o no pueden extraerse conclusiones directas, sobre las posibles relaciones con las prestaciones.

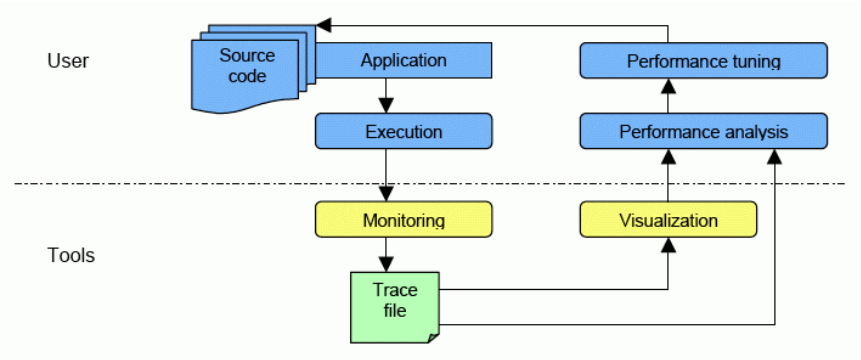

Figura 2.9: Ciclo de uso de una herramienta de visualización (de traza)

El análisis de prestaciones implica muchas fases complicadas y consumidoras de tiempo. En un principio el desarrollador parte de una serie de hipótesis sobre las prestaciones, debidas (si es el caso) al conocimiento de la aplicación, al modelo de prestaciones (que índices se utilizan, y cuando se suponen buenos o malos), y a las observaciones obtenidas en tiempo de ejecución. Con las herramientas de visualización de prestaciones, después de una instrumentación y monitorización, se llega a recoger los datos de las muestras de los datos de prestaciones e índices de las herramientas. De esta manera el desarrollador puede examinar esta información, y intentar probar (o no) sus hipótesis iniciales sobre las prestaciones, y pensar maneras de mejorar las causas de perdida de prestaciones observadas.

De este punto, se observa que el análisis de prestaciones basado en herramientas visuales, consume un importante tiempo al desarrollador, que podría usarse en otras fases del proceso de desarrollo de la aplicación. Y que este tiempo tendría que poder evitarse, con la ayuda de herramientas que automatizasen el proceso del análisis de prestaciones a partir de los datos obtenidos durante las fases de instrumentación y monitorización.

En las herramientas automáticas, como veremos en la siguiente sección se intenta automatizar esta fase de análisis. La herramienta automática intenta inferir las causas de los problemas de rendimiento, e informa de su presencia, y en ciertos casos también hay un retorno al usuario de medidas que puede tomar para mejorar las prestaciones de su aplicación.

Estas herramientas utilizan la información generada en la fase de monitorización, no solo para mostrar dicha información (en determinadas herramientas no se utiliza directamente la visualización), sino que se incorporan herramientas (o tareas) adicionales de análisis (y en algunos casos posterior sintonización automática), tanto estáticas (basadas en técnicas postmortem) como en dinámicas, para encontrar causas a ciertos tipos de problemas de rendimiento y proponer soluciones.

Capítulo 3

Análisis automático de prestaciones

En este capítulo examinamos la aproximación del análisis automático de prestaciones, desde una perspectiva histórica, y de las diferentes metodologías utilizadas. De estas ultimas incidiremos en el caso de unas cuantas herramientas representativas disponibles.

Aunque en el campo de prestaciones de las aplicaciones paralelas se han producido diferentes hitos durante las ultimas décadas, la situación actual no es todo lo buena que cabria esperar, aún hay lagunas significativas en las herramientas software que permitan soportar las fases del análisis de prestaciones de una forma flexible, y en que se puedan automatizar todos los pasos posibles.

Desde hace unos diez años han venido surgiendo diferentes aproximaciones sobre como enfocar el análisis automático de prestaciones de sistemas paralelos y/o distribuidos, que iremos desarrollando durante este capítulo.

La progresiva complejidad de los sistemas físicos (tanto en supercomputadores como a nivel de clusters), así como la cada vez más compleja configuración de elementos software, tanto de sistema, como de librerías, y aplicaciones realizadas, ha hecho quedar obsoletos (o poco útiles) muchos de los mecanismos de visualización gráfica de prestaciones utilizados con anterioridad.

Básicamente aparecieron una serie de problemas relacionados con:

- Los niveles de software, su número y su complejidad, hace muchas veces inviable un conocimiento profundo de todas las técnicas para la mejora de las prestaciones.
- El conocimiento, visual, de la existencia de un problema no se traduce necesariamente en saber las causas de lo que ocurre y como hay que solucionarlo.
- La escalabilidad de las soluciones: No es sencillo analizar las implicaciones observadas en prestaciones de forma visual para sistemas con decenas o centenares de nodos.

Automatizar el análisis de prestaciones, significa la automatización del proceso de obtener información de las causas que han provocado el comportamiento ineficiente de la aplicación, con una aproximación basada en intentar reducir el tiempo y la cantidad de trabajo que se dejaba en manos del desarrollador. En particular supone fases donde habrá que identificar los problemas de prestaciones observados, clasificarlos por tipos y importancia, y relacionarlos con el código fuente, para indicar aquellos puntos donde sea adecuado (o posible) actuar para minimizar (o eliminar) las causas que los provocaron.

En un análisis automático de prestaciones idealizado (ver figura 3.1), el usuario comienza un ciclo de análisis [Cou93, Wol03], a partir de su código de aplicación, le añade instrumentación (ya sea de tipo profiling o trazas) intentando reducir la perturbación, ejecuta la aplicación, y durante el tiempo de ejecución se recolecta datos de prestaciones, que bien en el mismo momento o a posteriori son introducidos en la herramienta de análisis de prestaciones. La herramienta ha de ayudar al usuario a identificar y corregir los problemas de su código. El proceso se repite hasta que el usuario obtiene las prestaciones deseadas de su aplicación, normalmente a partir de algunos requerimientos iniciales que se habían impuesto.

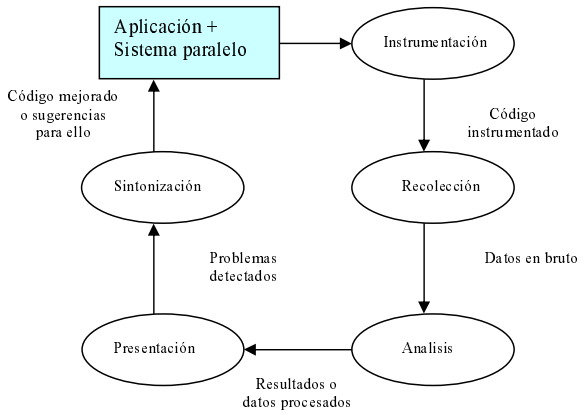

Figura 3.1: Flujo ideal del análisis de prestaciones

En herramientas reales, no existe ninguna que use este flujo de trabajo idealizado, en lugar de esto las herramientas intentar soportar parte del flujo, sopesando la funcionalidad que se quiere conseguir con los overheads creados por la perturbación, y la usabilidad final ofrecida al usuario. Las herramientas reales tienen que tomar decisiones sobre el tipo de datos que recolectaran, y se puede necesitar realizar un análisis de prestaciones con información parcial o imperfecta. Por tanto los datos accesibles, y el proceso de análisis pueden (y a veces deben) ser utilizados y/o realizados en diferentes fases de este ciclo de sintonización de la aplicación, en lugar de momentos, o etapas concretas como en las etapas ideales de la figura 3.1.

3.1. Conceptos

En estas herramientas se plantea, ir más alla del ciclo clásico que incluye como método principal la visualización de datos de monitorización. El siguiente paso introducido es el análisis de estos datos para sacar conclusiones sobre que está pasando con las prestaciones, y el lugar (temporal y en código), donde aparecen los problemas. Y posteriormente como hay que sugerir actuar para reducir los efectos de estos problemas, o incluso eliminarlos.

En las herramientas de análisis automático de prestaciones existentes, se presentan diferentes estrategias a la hora de dar soporte a estos requerimientos:

- Aplicación de la herramienta en fase Post-mortem, on-line, o preejecución: En fase post-mortem la herramienta (o complementos a esta) deben recolectar la información completa durante la ejecución de la aplicación, para aplicar la herramienta sobre esta información posteriormente a la ejecución. En las basadas en fase on-line, la información se recolecta y analiza durante el tiempo de ejecución. En herramientas basadas en preejecución se realizan todos los análisis antes de que el programa sea ejecutado.

- Manual o Semi-automática: Los análisis y la detección de los problemas pueden automatizarse de diferentes maneras, mientras algunas pueden confiar parcialmente (o totalmente) en el usuario para interpretar los resultados, y sintonizar la aplicación.

Esta clasificación no es totalmente ortogonal, y las diferentes herramientas pueden utilizar combinaciones de estas estrategias.

En la discusión posterior no consideraremos las estrategias preejecución, ya que se basan en metodologías de simulación previa, o estimación de las prestaciones por medio de modelos [Bro00] de las aplicaciones paralelas, técnicas por otra parte con tendencia a errores en caso general, sino se conoce algún modelo previo de la estructura o implementación de la aplicación, y se caracterizan de forma adecuada la aplicación junto con el modelo de la arquitectura de destino de ejecución [Sna02]. Estas técnicas son basadas bien en la creación de modelos del paradigma de programación [Sin97], o bien mediante simulación [Del97, Del97a, Bag98, Bag01] por medio de simulación discreta por eventos ([Pll05]) o continua en el tiempo, o métodos basados en las técnicas Montecarlo. Otra una técnica bastante utilizada es la de simulación guiada por traza [Yan96, Gir00], mediante modelos de simulación del cómputo y de las comunicaciones [Hey97, Sin97, Pra00, Vad01, Vad04], o en algunos casos en detección de estructuras sin previos conocimientos [Blo95]. Tecnicas similares se emplean con el análisis de escalabilidad de las aplicaciones mediante simulaciones, para comprobar si las prestaciones iniciales pueden escalar correctamente (vease SCALA [Sun02] o [Mal94]).También con las diversas técnicas de simulación de ejecución de las aplicaciones, se pueden emplear comparaciones con ejecuciones reales donde se obtienen valores para optimizar [Vud04] los modelos [Mid04] realizados a partir de código fuente. En algún

caso [Li96] también se propuso trabajar con el código fuente de forma directa, de manera que se creaba una representación de este y se intentaba corresponder con una base de conocimiento de buenas metodologías orientadas a prestaciones. En algún caso reciente [Kuh04], mediante analizadores estáticos de código fuente, se ha propuesto predecir prestaciones con modelos de tiempos asociados al paso de mensajes (ver sección 2.4), y las primitivas punto a punto y colectivas. Pero siempre hay algunos factores como las latencias de red, contención y ancho de red, difíciles de simular/predecir a priori.

Respecto a las herramientas otro factor relevante es la escalabilidad de la estrategia usada. Las que trabajan de forma adecuada con sistemas pequeños, no tienen porque trabajar adecuadamente en sistemas más grandes, una determinada estrategia que tenia unos overheads concretos puede fallar o verse inutilizada para sistemas grandes. Adicionalmente, aquellas técnicas que confíen en el usuario para algunas (o la mayor parte de)tareas manuales serán muy ineficientes ya que el usuario no sera capaz de captar toda la información disponible e medida que el tamaño del sistema crece. Como era el caso, con los sistemas clásicos de visualización comentados.

Otra idea común, en la mayoría de las herramientas automáticas (o semi-automáticas) es la utilización de una aproximación basada en una base de conocimiento, para identificar los problemas de prestaciones presentes en las aplicaciones. Este tipo de herramientas disponen una base de conocimiento de problemas de prestaciones que son usada para poner en correspondencia con las observaciones obtenidos (habitualmente) de los ficheros de traza (aunque también es aplicable en profiling) obtenidos de la ejecución de las aplicaciones. Cuando los problemas son detectados, se incluyen diferentes estrategias en los métodos de análisis para indicar como solucionar o evitar el problema con información contenida en la base de conocimiento.

3.2. Modelos del análisis

Ampliaremos en esta sección la discusión sobre las diferentes aproximaciones [Mar04] para la elaboración de herramientas de análisis automático de prestaciones, y las ventajas y inconvenientes de cada una.

Para esta discusión utilizaremos la siguiente nomenclatura de las diferentes aproximaciones disponibles:

- Análisis automático de prestaciones estático: El análisis se realiza en una fase post-mortem y puede considerarse toda la información detallada recolectada (normalmente por monitorización mediante traza) durante la ejecución de la aplicación.

- Análisis automático de prestaciones dinámico: Utiliza métodos dinámicos para introducir instrumentación durante la ejecución, con el fin de evitar el uso de trazas, y poder controlar la cantidad de instrumentación insertada en

la aplicación. Esta aproximación es útil para aplicaciones de larga duración con comportamiento estable.

- Sintonización automática de prestaciones dinámica: Intenta adaptar la aplicación durante la ejecución, sin pararla, recompilar y volver a ejecutar. Aproximación en especial útil para aplicaciones con comportamiento dinámico.

En la estrategia estática con fase post-mortem, la herramienta es aplicada después de la ejecución de la tarea, y usa la información recopilada durante la ejecución de la aplicación, usualmente mediante técnicas de traza. De facto esta estrategia es una de las más habituales en las herramientas de prestaciones de soporte al modelo de paso de mensajes. Como vimos en las secciones dedicadas a tracing en el capítulo anterior, existen diferentes técnicas para disminuir el nivel de perturbación provocado (buffers de traza, traza selectiva, etc.), y otro punto importante es que dejamos fuera las fases de análisis y de sintonización de la herramienta, del tiempo de ejecución de la aplicación, eliminando la perturbación que pudieran causar. Otro punto relevante, es que en el modelo de paso de mensajes, el paralelismo es explicito, y por tanto con esta estrategia post-mortem, la mayor parte del comportamiento de la aplicación puede se capturado (o extrapolado) a partir de los eventos de traza que pueden contener todos los elementos (como las primitivas) del modelo de paso de mensajes utilizadas.

Debido a que el tiempo, en la aproximación estática, no es crítico, y la información disponible es muy detallada (debido a la disposición de todo el paralelismo explicito en la traza), es posible realizar un análisis, que intente en primer termino identificar los problemas de prestaciones, determinar sus causas, relacionarlas con el código fuente de la aplicación, y finalmente pueda sugerir ciertas recomendaciones al usuario. Como desventajas de la aproximación estática, hay que tener en cuenta que:

- Para poder realizar un análisis completo, el fichero de traza debe contener el máximo posible de información de la ejecución de la aplicación. Esto requiere que se puedan instrumentar el máximo posible de eventos de la aplicación (en nuestro caso de paso de mensajes, el mayor número de llamadas a primitivas, y/o otras que el desarrollador de la herramienta considere oportunas). En algunos casos, esta instrumentación, puede introducir una perturbación substancial en la ejecución de la aplicación, o incluso ocultar (o modificar) su comportamiento real.

- La cantidad de datos generada durante la monitorización por traza de una aplicación puede ser muy elevada. Los ficheros de traza pueden llegar a contener varios gigabytes de datos, y ser díficiles de almacenar y procesar. A pesar de esto, cabe tener en cuenta que en muchos casos, las aplicaciones de larga ejecución exhiben un comportamiento regular, usualmente conteniendo iteraciones, en las que en forma estable, puede ser suficiente analizarlas por separado, ya que se repetirán múltiples veces en la traza.

- El análisis que se produce, está basado en una instancia concreta de la ejecución de la aplicación, las recomendaciones y sintonización que se puedan realizar son adecuadas para esa ejecución. Ciertas aplicaciones pueden cambiar su comportamiento en diferentes ejecuciones, o a lo largo de su misma ejecución. En estos casos, las modificaciones sugeridas por el análisis post-mortem pueden ser insuficientes para cubrir el comportamiento dinámico de la aplicación.

Teniendo en cuenta estos parámetros, la aproximación es aconsejable para aplicaciones no excesivamente grandes (o bien grandes con estabilidad de comportamiento), teniendo cierta perseverancia durante diferentes ejecuciones, o en durante repeticiones de la misma ejecución. En estos casos, la cantidad de datos no sera excesiva, y podrá realizarse un análisis, para proporcionar recomendaciones útiles para la mayoría de las ejecuciones de la aplicación. Para otros casos que incumplan parte de estas premisas, podemos usar diferentes técnicas, que permitan superar las dificultades de una aproximación estática, ya sea mediante técnicas de reducción de cantidad de la información en traza, mediante técnicas compresión, filtraje selectivo, resumen de estados, etc. O bien mediante técnicas de gestión de datos de múltiples experimentos de ejecución de la aplicación. Este conjunto de técnicas puede permitir la aplicabilidad de estas herramientas a un amplio rango de aplicaciones.

Precisamente en este último punto, en la ejecución de una instancia de la aplicación, es crítica habitual de la aproximación estática (post-mortem), es que tanto útiles son las informaciones obtenidas para el comportamiento global de la aplicación. Normalmente el análisis, de las herramientas estáticas, se realiza sobre una o más ejecuciones de la aplicación para intentar optimizarla sobre el sistema concreto, y sobre un conjunto de datos habitual. La cuestión que se plantea es que tanto útiles son los datos del análisis para otra ejecución de la aplicación con variaciones de su entrada o de su entorno de ejecución. La respuesta a este tema no es clara, pero hay que tener en cuenta que la mayoría de las herramientas estáticas se han diseñado para la evaluación de prestaciones de aplicaciones bajo el comportamiento ẗabitual¨(estadísticamente normal o uniforme), de manera que se supone que se aplica a unas aplicaciones con un comportamiento homogéneo frente a variaciones de los datos (o sea con poca sensibilidad frente a variaciones de la entrada). En los casos que esto no suceda, se han propuesto diferentes aproximaciones a los entornos multiexperimento [Wol03, Klu05a], de manera que sea posible correlacionar datos obtenidos de diferentes ejecuciones, o bien usarlos como históricos de comportamiento. También es posible en estos casos, utilizar las ejecuciones en entorno de múltiples versiones del código original, para tratar de evaluar, las mejoras o perdidas producidas por diferentes actuaciones sobre el código fuente. O incluso estudiar la escalabilidad de la aplicación, bajo cambios del sistema de ejecución [Klu05a].

La segunda aproximación a las herramientas automáticas de análisis, es basarlas en información dinámica [Mar04], con objetivo principal de reducir el total de

datos almacenados, y/o limitar la perturbación en la aplicación debida a la monitorización. En esta aproximación, la aplicación es monitorizada mediante instrumentación dinámica [Mil95, Buc00, Cai00], y diagnosticada, durante su ejecución. Los datos recogidos son usados durante una fase on-line de análisis, para determinar los problemas más relevantes, y correlacionarlos con las tareas de la aplicación, y los módulos o funciones del código, y intentar explicar las razones al desarrollador. Esta aproximación ofrece algunas ventajas:

- El proceso de análisis suele definir y procesar las medidas de prestaciones que hay que tomar en tiempo de ejecución, no se usan habitualmente trazas. La instrumentación puede añadirse de forma selectiva de forma dinámica, en función del comportamiento del problema, o que esta instrumentación se active cuando se detecten ciertos valores de rendimiento (o de falta de el). Así podemos controlar o reducir la perturbación creada.

- El análisis on-line puede enfocarse en aspectos específicos de la ejecución (los problemas más significativos), seleccionando y refinando sus medidas en función de los resultados que se vayan obteniendo, o basándose en la historia de la ejecución. Esto permite disminuir el total de datos de medida, focalizandolos su obtención donde sea necesario mayor detalle.

- Los problemas pueden identificarse de una manera mucho más rápida que en una aproximación post-mortem. De hecho *deben* identificarse rápidamente para no perjudicar la ejecución de la aplicación.

- El análisis de hecho puede ejecutarse on-line pero usando los recursos de una maquina (o nodo paralelo) separada, sin introducir overhead extra en la ejecución de la aplicación, excepto para la limitada instrumentación, y las transferencias de datos sobre la red. Aunque hay que tener en cuenta la sincronización de datos en el análisis, con el comportamiento actual de la aplicación.

La aproximación dinámica es mejor para programas iterativos, que realicen una larga ejecución con grandes volúmenes de datos. Pero igual que en el caso de la aproximación estática, su análisis está basado en una sola ejecución de la aplicación. Además en este caso cuando cuando el comportamiento de la aplicación depende de la iteración de la ejecución o de los datos de entrada, las recomendaciones sugeridas pueden ser inadecuadas para una posterior ejecución de la aplicación.

Por último la aproximación de sintonización dinámica [Hol93, Mor03a, Vud04], intenta complementar las otras dos aproximaciones en el rango de aplicaciones con una variabilidad de comportamiento dinámico significantivo durante la ejecución. En esta aproximación se analizan los resultados obtenidos con técnicas de profiling (habitualmente) o traza, se toman las decisiones adecuadas y se intenta modificar el código ejecutable del programa paralelo durante un tiempo de ejecución de éste sin, o con mínima, intervención del usuario. La sintonización dinámica tiene la ventaja de que no se necesita volver a compilar ni enlazar los ficheros fuentes, ya que los

cambios se realizan directamente sobre el ejecutable. Como desventaja cabria citar, que para evitar la perturbación el análisis tiene que realizarse lo más rápido posible, y muchas veces con limitada información. También aquí podemos utilizar técnicas para minimizar la perturbación, como la focalización de las medidas sobre las zonas en que se produce la perdida de prestaciones, o otras de aumentar/disminuir la instrumentación en función de la marcha de los índices de prestaciones.

En esta aproximación, el desarrollador puede concentrarse en las fases de diseño y de desarrollo de su aplicación, y después la sintonización dinámica toma control de la aplicación monitorizando la ejecución, analizando el comportamiento y modificando la aplicación para mejorar las prestaciones. Cuando la aplicación varia de comportamiento, según los datos de entrada, o varia de una iteración a otra, el analizador detecta el comportamiento actual, y determina como adaptar la implementación a las condiciones cambiantes de la ejecución presente.

Otro problema con la sintonización dinámica es que, habitualmente, no se consigue una información útil de cara al usuario final, ya que su objetivo es adaptar el comportamiento de la aplicación al sistema final de cómputo, y estas herramientas no suelen ofrecer información final sobre los cambios realizados. Ya que estos dependen fuertemente de la aplicación y de los datos de entrada, y esta sintonización puede variar en diferentes puntos del programa, o a veces tener que deshacer cambios previos realizados en una fase previa de sintonización, que a dejado de ser útil (o es contraproducente) en la iteración o periodo de tiempo actual. En este sentido el usuario no obtiene el retorno de información necesario, que le pueda permitir evitar estos problemas de prestaciones a priori.

En todo caso seria posible, al modificarse el código original, sugerir aportar código o cambios en el, de manera que se informe a los usuarios de los cambios realizados, siempre que estos hayan obtenido una mejora de las prestaciones o de algún índice deseable de estas, por cierto periodo de tiempo o durante la ejecución global. Y que estos cambios no hayan estado invalidados o cambiados a posterioridad.

Además de estas consideraciones, la aproximación por sintonización dinámica presenta otros puntos a ser tratados:

1. La perturbación debe minimizarse. Además de la instrumentación semejante al caso dinámico, hay una serie de overheads extras relacionados con las comunicaciones del monitor, el proceso de análisis, y las modificaciones de código.

2. El análisis debe ser muy simple, y las decisiones deben tomarse en un tiempo corto para ser efectivas en la ejecución del programa. Sino el tiempo de actuación sobre el problema podría ser mayor que la duración de este.

3. Las modificaciones no deben envolver un alto grado de complejidad, ya que no es asumible que cualquier tipo de modificación en cualquier aplicación en cualquier entorno pueda realizarse durante su ejecución. Los mecanismos de alteración de código son fuertemente dependientes de la plataforma

hardware, y de software de sistema utilizados.

4. Se necesita conocimiento de la aplicación, sobre que medidas hay que tomar, donde tomarlas, y que puede modificarse y cuando. Esto puede realizarse básicamente a partir de dos casos: a) Un conocimiento exhaustivo de la aplicación, definiendo estos puntos; b) Mediante el uso de frameworks de creación de las aplicaciones [Ces02, Ces99, Par98], donde actuamos sobre estructuras predefinidas (que el desarrollador usa) donde conocemos sus puntos de medida y actuación.

Por todas estas razones, la evaluación y las modificaciones realizadas no pueden ser excesivamente complejas. Debido a que la sintonización se ha de realizar en tiempo de ejecución, es difícil realizarla las diferentes fases sin un conocimiento previo de la estructura y funcionalidad de la aplicación. Si el desarrollador puede desarrollar cualquier tipo de aplicación, los potenciales problemas pueden tener un espectro muy amplio, y no tener una clara definición en los términos de puntos de observación y medida, haciendo extremadamente difícil o incluso imposible una aproximación por sintonización dinámica.

3.3. Herramientas automáticas

Basándose en las ideas de prestaciones expuestas, y en las dificultades de resolver los problemas del análisis de prestaciones, mediante herramientas de visualización y monitorización. Diversas ideas, durante la última década se han convertido en herramientas de análisis automático de prestaciones. Intentando automatizar el proceso cíclico de análisis en mayor o menor medida, estas herramientas has supuesto el aporte de ideas a nuestro trabajo en algunos casos, en otros han sido visiones complementarias al mismo problema, y en el caso de la primera versión de KappaPI, la idea inicial desde la que partimos.

Resumimos aquí, de forma breve, algunas de las aproximaciones, y observaremos con más detalle, en las próximas secciones otras herramientas de las que creemos más destacables sus aproximaciones, y su difusión en la literatura asociado a análisis automático de prestaciones.

Los trabajos de P.C.Bates [Bat86, Bat95], proponen un lenguaje denominado EDL (Event Definition Language) para especificar comportamientos ineficientes, basados en descripciones declarativas de eventos presentes, utilizando expresiones regulares, pudiéndose combinar y/o unir las declaraciones mediante operadores de mayor nivel. El objetivo es el análisis de la aplicación para la depuración para detectar comportamientos erróneos. Sus trabajos son de los primeros que proponen declaración de eventos para problemas en las aplicaciones paralelas. Uno de los problemas de EDL, es el estar basado en expresiones regulares, creando así una especificación basada en autómatas finitos, no pudiendo manejar problemas de prestaciones basados en estados.

En el trabajo del análisis de ciclos perdidos (Lost cycles analysis) [Cro94, Mei95], se utilizaba una técnica de análisis basada en muestreo del estado de la aplicación para determinar cuando que se estaba realizando trabajo útil o cuando no. Se requería una instrumentación de código que habitualmente se realizaba por la herramienta, o bien por el usuario, que introducía indicadores de estado sobre la aplicación. Cualquier tiempo que la aplicación pasaba no estando directamente relacionado con la computación se calificaba como de ciclos perdidos. Estos ciclos perdidos eran divididos en una serie de categorías: desbalanceo de carga, paralelismo insuficiente, perdidas en sincronización, perdidas en comunicación, y contención de recursos. Los ciclos globales de la aplicación se clasificaban en función de estas categorías, a medida que se acumulaban los ciclos perdidos encontrados. Por ejemplo si existían ciclos de trabajo en una tarea, mientras en las otras esperaban por sincronización, entonces es que existía un desbalanceo del trabajo. Una de las ventajas de este análisis es que permitía prever la escalabilidad de la aplicación, pero sufría del problema de las técnicas de muestreo, siendo difícil o casi imposible relacionar el análisis con el código fuente de la aplicación. Además, prácticamente cualquier información tomada de una aplicación en tiempo de ejecución (basada en técnicas de tracing y/o profiling) puede tomarse como un estado de la aplicación, y por tanto realizarse todos los análisis mencionados.

En Simple [Dau93, Moh93], y Mentor [Dau94] se introdujeron conceptos de herramientas guiadas por conocimiento del campo a analizar, y la necesidad de disponer de una descripción de este conocimiento a diferentes niveles de la arquitectura de las herramientas.

En Paradise [Kri96, Kri96a], fue de las primeras herramientas donde se introdujo el concepto de ciclo de búsqueda de prestaciones (parecido al mostrado en la figura 3.1). Y se incluía la idea de retornar al usuario recomendaciones de como mejorar su aplicación. También se estudiaban ciertas estructuras de los paradigmas de programación de las aplicaciones para detectar fases de comportamiento concreto (puntos de sincronización, puntos de iniciación, etc.) dentro de la aplicación.

En charm [Sin96], un lenguaje de programación paralelo con herramientas de análisis, existe la idea de detectar fases de la ejecución de la aplicación paralela, creando unos puntos de separación de fases lógicas, donde en cada fase eran computadas métricas sobre la granularidad del cómputo paralelo y las comunicaciones. Después para cada una de las fases se intentaba examinar el camino crítico hasta el momento, analizar donde había tiempo perdido, evaluar los desbalanceos detectados, y el comportamiento de los datos utilizados, presentándose al final sumarios de cada fase.

En Opal [Ger97], se desarrollo una aproximación basada en un sistemas de reglas, con refinamiento de hipótesis, de manera que se plantean una serie de ellas encadenadas, que a medida que son comprobadas se pasa a la siguiente hipótesis. Cada regla necesita una serie de informaciones de prestaciones que son las que se van monitorizando en cada momento, para controlar el total de información (y memoria para almacenarla), y usar/producir la información que sea necesaria en cada momento.

En Finesse [Muk99, Muk01], el análisis se centra en una búsqueda de overheads como medida de la perdida de prestaciones que sufre la aplicación. Mediante un análisis automático estático se produce la recolección de información de los overheads y su clasificación para detectar zonas donde se acumulen estos, y puede recomendar cambios o transformaciones de código a realizar.

Otro autor J. Vetter [Vet00], sugiere de técnicas de aprendizaje en las cuáles se clasifican los problemas de análisis automático de aplicaciones MPI con mensajes punto a punto. La idea es utilizar un entrenamiento de la herramienta (mediante técnicas de aprendizaje), bajo unos benchmarks, de manera que esta construya un árbol de decisión que luego utilice para el análisis de otras aplicaciones. Los benchmarks han estado preparados para que se demuestren cuan eficientes o ineficientes se han mostrado en el sistema estudiado, y poder crear un árbol de decisión, a partir del que se analizaran las aplicaciones posteriores. La utilización del árbol de decisión permite obtener una clasificación automática, y en cierta manera que explique también las causas por la que se clasifico así.

Kranzmuller [Kra00], utiliza análisis basado en grafos, para detectar errores en las construcciones de los programas paralelos. Se sigue una vista temporal de las tareas, con los eventos que suceden en la linea de tiempo, y relaciones entre ellos de que un evento paso antes que uno (o unos) determinados. Se pueden definir patrones que especifican el comportamiento erróneo de forma gráfica. Se realizan operaciones, por parte de la herramienta, de selección de comportamientos parecidos a los patrones, o de construcción de macrobloques, a partir de los elementos enlazados.

En la herramienta Autopilot [Rib01], destinada a control adaptativo en tiempo real, el análisis automático se realiza por captura de datos basada en sensores de prestaciones, los cuáles son disparados, mediante reglas en lógica fuzzy, cuando se cumplen unos limites o condiciones impuestas por el usuario, de manera que se proponen cambios en la gestión de recursos, mediante unos actuadores se intentan modificar aquellos parámetros sugeridos, por ejemplo en el sistema de ejecución.

En herramientas de visualización basadas en traza o profiling, también comienza a ser habitual (en los ultimo tiempos) la introducción de algunos elementos de detección de problemas, aunque no se produce análisis como tal, como mucho detectar algunas estructuras de programación ineficiente. Por ejemplo, en Vampir, se están introduciendo [Knu04, Knu05] algunas técnicas tentativas de detección de múltiples comunicaciones punto a punto, para reconvertirlas en operaciones colectivas.

En las siguientes secciones describiremos con más detalle algunas de las herramientas de análisis automático más significativas.

3.3.1. KappaPI

Desarrollada en la universidad Autónoma de Barcelona (UAB), KappaPI [Esp97, Esp98, Esp00] (Knowledge-based Automatic Parallel Program Analyzer for Performance Improvement), es una herramienta estática de análisis automático de

prestaciones, basada en análisis post-mortem de la aplicación, basándose en ficheros de traza y en la base de conocimiento que incluye varios de los principales problemas de las aplicaciones de paso de mensajes. La herramienta ayuda al usuario en el proceso de mejorar las prestaciones de su aplicación por medio de detectar los problemas principales de esta, inferir las causas de estos problemas y relacionar las causas con el código fuente de la aplicación.

En el paso preliminar, la aplicación PVM se ejecuta con la herramienta de monitorización TapePVM (trazador para entornos PVM), para obtener los ficheros de traza que luego serán analizados por la fase de análisis mediante la herramienta.

El comportamiento de la aplicación se analiza, en KappaPI, en dos partes:

a) A partir de la traza post-mortem de la aplicación, se realizan métricas simples para determinar los tiempos idle, donde uno o más procesadores (u nodos) no realizan cómputo, de manera que se consiga una visión general de las prestaciones de la aplicación por medio de medir la eficiencia de los diferentes procesadores del sistema. KappaPI considera, así, ineficientes en prestaciones aquellos intervalos de tiempo donde hay procesadores que no están realizando trabajo útil; por ejemplo permanecen bloqueados esperando por algún mensaje. La eficiencia, es entonces, considerada como porcentaje de tiempo donde se está realizando trabajo útil. Cuando aparecen intervalos de tiempo no usados (idle), estos intervalos han de suprimirse o optimizarse para mejorar las prestaciones de la aplicación. En este primer paso se le puede proporcionar al usuario, información (en forma de índices de porcentaje) sobre el comportamiento global de la aplicación, pero sin ideas de los problemas de prestaciones y sus causas.

b) Después, basándose en la lista de tiempos idle, se inicia un proceso de inferencia, basado en reglas asociadas a cada problema de prestaciones. Este proceso de inferencia a traves de reglas permite deducir nuevos hechos que permiten subir de nivel de abstracción hasta refinar y llegar al problema de prestaciones causante de la ineficiencia. Este proceso se repite mientras haya conjuntos de reglas aplicables, hasta que no podamos deducir nuevos hechos. Finalmente a partir de las causas encontradas, y un análisis de las zonas de código fuente relacionadas con la causa, se emiten sugerencias sobre posibles mejoras a realizar sobre el código, para evitar/eliminar las causas que producían la disminución de prestaciones.

Este análisis se aplica a la traza del programa, en ciertos pedazos de estas (denominados chuncks) para manejar correctamente los problemas de escalabilidad de las trazas (al ser estas de tamaños elevados de almacenamiento). El análisis se aplica progresivamente según estos chunck de manera que lleguemos, a ciertas conclusiones de los problemas presentes, y estos son acumulados y clasificados por repeticiones que se encuentren al largo de la aplicación.

En la figura se puede observar una salida de la herramienta, relacionando una sugerencia con las posiciones en el código relacionadas, y las actuaciones para disminuir o evitar la ineficiencia. En el caso de la figura, se ha realizado una detección de un esquema Master/slave, donde se observa un número incorrecto de workers en la aplicación para el trabajo a realizar.

3.3.2. Expert

En la herramienta Expert [Wol00, Wol02, Wol04] que forma parte de un proyecto más amplio (ver figura 3.2) denominado KOJAK [Moh01, Moh03a]. El objetivo es análisis automático de prestaciones, en entornos híbridos SMP, basándose en programación de MPI, OpenMP o híbrida entre las dos. Se toma una aproximación basada en base de conocimiento de problemas, especificados mediante lenguajes de script (python, tcl) que permiten incorporar nuevos problemas a la lista de disponibles para la herramienta. Aunque en las ultimas versiones de la herramienta se ha trasladado a clases C++ y Python [Bha05].También se utiliza una API denominada EARL [Wol99, Wol01] que ofrece abstracciones de los datos y las operaciones típicas a realizar con las trazas utilizadas por la herramienta. EARL es utilizado en la especificación de los problemas, para realizar operaciones con la traza, como relacionar eventos, buscarlos, o identificar sus parámetros. Básicamente mediante el uso de la especificación, junto con EARL se permite añadir capacidades de detección para nuevos problemas de prestaciones.

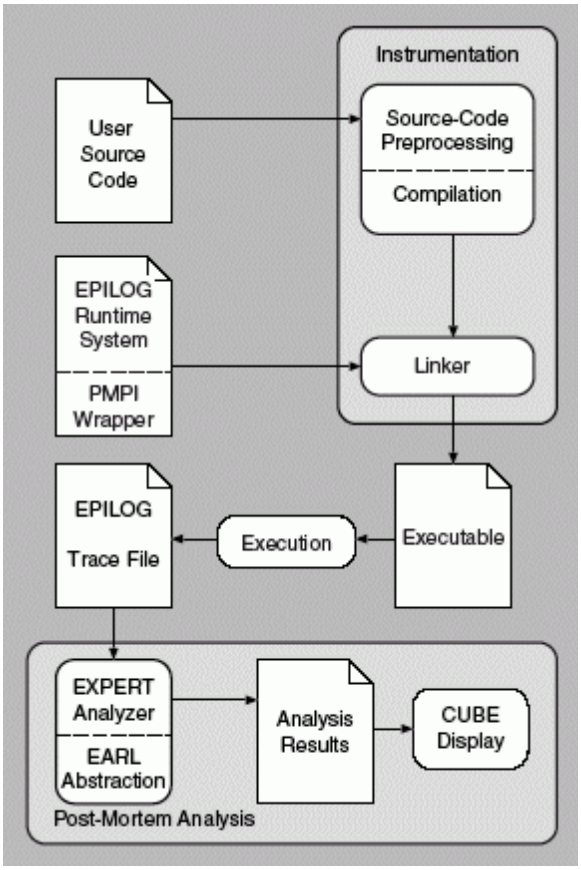

Figura 3.2: Arquitectura del sistema KOJAK

El sistema completo KOJAK (3.2) es una herramienta de tipo post-mortem basada en traza generada por medio de wrappers [Bha05]. Se instrumenta el código fuente mediante el interfaz PMPI (que intercepta llamadas MPI), y el uso de la librería de traza EPILOG (formato de traza utilizado) que se enlaza con la aplicación. En la ejecución se generan los ficheros de traza, incluyendo información de las llamadas MPI punto a punto y colectivas, y información de OpenMP (basada en instrumentación insertada por el compilador). Después de la terminación, se realiza post-mortem el análisis automático basado en los patterns antes mencionados de EXPERT, y se generan resultados para su visualización en CUBE (ver figura 3.3), otra herramienta de visualización que muestra los resultados de prestaciones en una vista tridimensional que incluye: 1) Métricas de prestaciones, o cuáles son las ineficiencias más grandes y de que tipo. b)Árbol de llamadas, donde se gasta el tiempo, en que rutinas. 3) Árbol de sistema, en que tareas se detecta el mayor consumo de tiempo. Todas las vistas incluyen coloración de los elementos de manera que se indican los porcentajes totales sobre la vista.

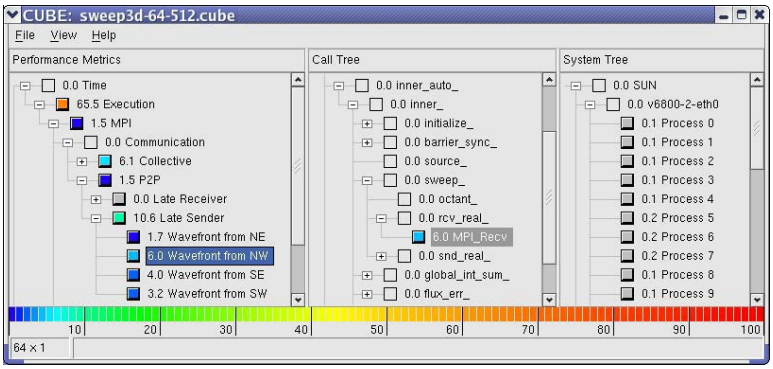

Figura 3.3: Visualización de los problemas mediante CUBE en KOJAK

Recientemente [Wol04] se ha añadido a Expert, la idea del refinamiento progresivo en la búsqueda de problemas por medio de compartir información común a los problemas, basándose en el trabajo previo [Esp00] en la primera versión de KappaPI, donde estos eran inferidos por progresivo refinamiento de sistemas de reglas hasta encontrar el problema especifico. Esta idea también es usada en la versión actual de KappaPI2 [Jor02, Mar04, Jor05], mediante el árbol de decisión de eventos (ver sección 4.7), en el cual aparecen primero los problemas menores, que a medida que aparecen más eventos pueden refinarse a problemas más específicos, que contienen al primero, o que son una combinación del primero más otros. En el caso de Expert, es visto como una especialización de clases, que compartiendo la misma información, puede instanciar uno u otro problema.

3.3.3. Scalea

En el proyecto Scalea, la herramienta de análisis automático de prestaciones utiliza JavaPSL [Fah01], que básicamente intenta especificar las propiedades de las prestaciones, utilizando el concepto de clases abstractas en el lenguaje Java. La idea es mostrar todo el conjunto de propiedades de prestaciones como una jerarquía de clases, utilizando los conceptos de orientación a objetos, y en este caso particular de Java, mediante polimorfismos, clases abstractas y reflexión. En particular mediante conceptos de herencia de manera que se base la definición de nuevas propiedades, en la especialización o composición de las ya existentes, para así integrar fácilmente las nuevas que se puedan definir. Durante la realización del análisis, se anotan en las clases todos los datos que vayan apareciendo de localización (en código) y de tiempos referidas a las propiedades concretas que se encuentren, creando instancias de las clases abstractas y anotando los datos en ellas.

Scalea automáticamente clasifica problemas de prestaciones por medio de identificar regiones de código done la cantidad de tiempo que es consumida por la aplicación en comunicación, y la relaciona con el código fuente usando vistas en árbol. También soporta comparación de diferentes ejecuciones directamente por medio de almacenar los datos de las ejecuciones en una base de datos, y soporta instrumentación dinámica dependiendo del soporte del compilador.

Scalea, junto con el lenguaje JavaPSL, se han integrado juntos en el nuevo entorno denominado Aksum [Fah02] (ver figura 3.4), que proporciona también soporte para múltiples experimentos de la misma aplicación. En Aksum el usuario puede seleccionar las propiedades de prestaciones a usar (o incorporan nuevas), incorporando datos de limites temporales de cuando una propiedad se considera crítica o no. Aksum instrumenta las regiones de código para recoger los datos basándose en las propiedades definidas. Se incorporan heurísticas para limitar las búsquedas a limites temporales, también especificados por el usuario. Aksum como resultado final, presenta las propiedades detectadas durante el proceso de búsqueda junto con las regiones de código que las causan.

3.3.4. Paradyn

El proyecto Paradyn [Mil95], de las universidades de Wisconsin y Maryland, ha seguido la idea de desarrollar un análisis automático de prestaciones de tipo on-line, basándose en la instrumentación dinámica (tambien pueden verse en otros trabajos como [Gu98]). En su idea central de modelo de prestaciones, se ha aplicado el concepto de motor de búsqueda de problemas basado en lo que denominan W3 (*Why, Where and When?*), en el que intenta analizar la aplicación de forma on-line, a partir de datos de profiling, durante su ejecución siguiendo un análisis multidimensional, entre los problemas de prestaciones que van apareciendo, los recursos afectados, y el tiempo.

La idea es realizar una búsqueda jerárquica partiendo desde una serie de métricas raíz que se van acumulando durante la ejecución, para centrarse en aquellas que

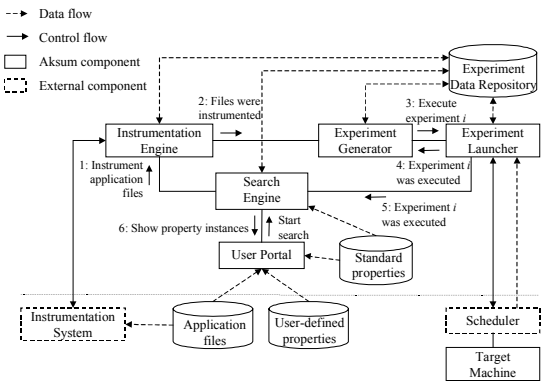

Figura 3.4: Entorno de ejecución de la herramienta Aksum

producen mayores resultados (superan determinados limites prefijados). A medida que se refinan las métricas, se establecen las localizaciones en el código que afectan a la métrica. De manera que dinámicamente se busca la localización en que se produce mayor incremento, para centrar el análisis, y determinar donde y como se está causando el problema. Se realiza un análisis de tipo *top-down*, en el cual se parte de la raíz de una hipótesis de prestaciones, con los tres ejes de W3, se refina la hipótesis a medida que se avanza la ejecución hasta llegar a la conclusión.

En el análisis multidimensional se utilizan: a) Las métricas que suelen suele ser un determinado porcentajes (de tiempo de CPU, de tiempo de bloqueo en comunicaciones), un determinado ratio (por ejemplo, operaciones por tiempo), o un valor simple como el número de procesadores que están computando. b) Los recursos del programa en forma de uso de los nodos, del disco, de red, colas de mensajes, etc. c) Temporal, para dividir la ejecución entre una serie de fases, en las que acumular y observar los resultados.

En la búsqueda bajo W3, Paradyn comienza con una hipótesis simple que va refinando a medida que obtiene la información en tiempo de ejecución (ver figura 3.5), donde se intenta identificar el tipo de problema de prestaciones (cuestión why?), donde ocurre este problema en la aplicación (cuestión Where?), y el tiempo donde ocurre (cuestión when?).

Se utiliza un esquema basado en daemons, en forma de arquitectura cliente/servidor, donde cada daemon envía la información a un daemon centralizado principal. El daemon puede pedir que se instrumenten partes de la aplicación en cada tarea, y de esta manera controlar el nivel de instrumentación introducido, así como la focalización en determinadas características o localizaciones. También se han incorporado mecanismos [Ber04, Hol98] de examinar el grafo de llamadas (call graph) y la pila de llamadas (call stack), permitiendo así examinar partes de la ejecución de la aplicación relacionadas con los problemas que estan surgiendo.

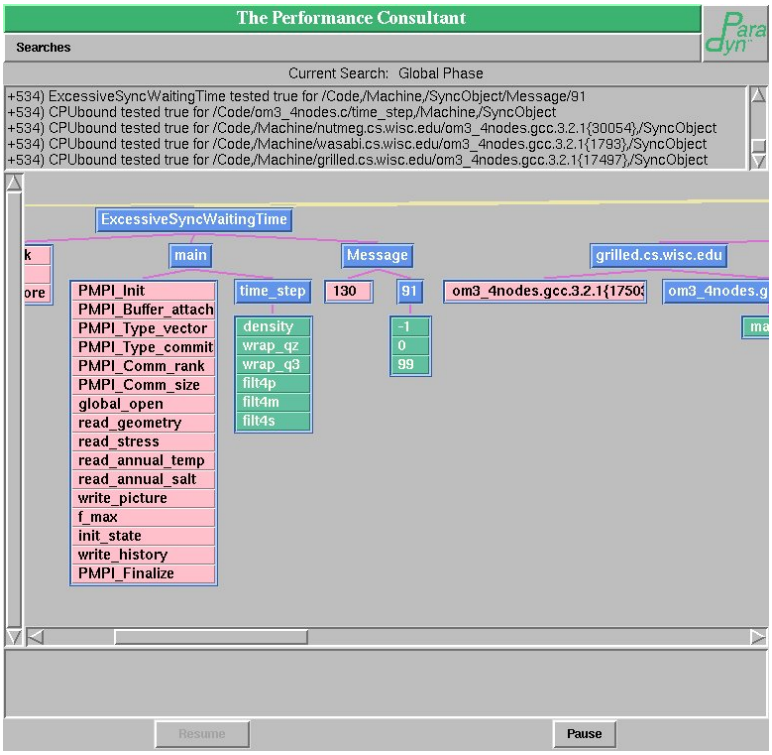

Figura 3.5: Busqueda realizada por el Performance Consultant en Paradyn

3.3.5. MATE

Esta herramienta [Mar04] desarrollada por nuestro grupo, en la universidad Autónoma de Barcelona, se enfoca hacia el análisis automático de prestaciones por mecanismos dinámicos. La herramienta MATE (*Monitoring, Analysis and Tuning Environment*), proporciona análisis automático de prestaciones con una estrategia dinámica. MATE realiza sintonización dinámica en tres fases básicas (y continuas): Monitorización, análisis de prestaciones y realización de modificaciones. Este entorno dinámicamente de forma automática traza e instrumenta una aplicación en ejecución para determinar su comportamiento. La fase de análisis realiza la búsqueda de problemas de prestaciones, detecta posibles causas y da solución mediante la actuación dinámica sobre el código de la imagen binaria en ejecución. Sin necesidad que la aplicación vuelva a ser compilada, enlazada, y ejecutada de nuevo.

La herramienta MATE se aplica a entornos PVM, y utiliza instrumentación dinámica por medio de la libreria DyninstAPI [Buc00, Hol02]. El entorno de sintonización se compone de tres componentes principales que cooperan entre ellos, controlando la ejecución de la aplicación (ver figura 3.6): 1) Un conjunto de módulos distribuidos encargados de recoger la traza; 2) Un módulo global del Analiza-

dor; 3) Un conjunto distribuido de módulos de sintonización.

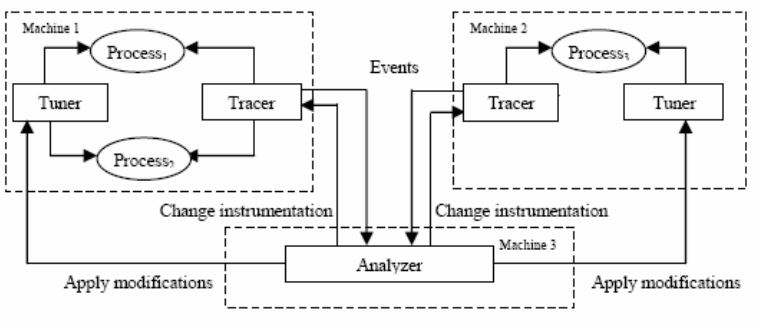

Figura 3.6: Entorno de sintonización dinámica en MATE

En los módulos de traza, nos encargamos de recoger los eventos producidos durante la ejecución. Estos módulos están presentes en todos los nodos donde hayan tareas de la aplicación están ejecutándose. La información se obtiene por instrumentación dinámica de las tareas, con datos sobre que, cuando y donde se ha producido el evento, y se recoge centralizadamente en el módulo de Analyzer. Los puntos de instrumentación, generalmente son especificados por el usuario, antes de la ejecución de la aplicación. Los puntos pueden variar durante la ejecución bajo demanda. Si el Analizador necesita más o menos información puede notificar al módulo de Tracing que varíe la instrumentación.

El módulo Analizador es responsable del análisis automático de prestaciones on-line. Recibe los eventos continuamente, examina los eventos según una ventana temporal, detecta los problemas, determina las causas y decide que hay que sintonizar en la aplicación. Cuando determina una actuación la envía al módulo de Tuner para que realice los cambios necesarios en el lugar y momento adecuados.

Los módulos de Tuner, automáticamente cambian la ejecución de la aplicación, por medio de insertar modificaciones en los procesos en ejecución. Se manipula directamente la imagen del proceso en memoria, no necesitan disponer de acceso al código fuente, o medios para reiniciar la aplicación. Todas las modificaciones se realizan sobre puntos definidos previamente (puntos de Tunning), y se realizan técnicas como modificar parámetros particulares en la aplicación [Mar04, Gio99], reemplazar llamadas a funciones con llamadas a diferentes implementaciones, insertar código adicional, etc. Estos cambios realizados por el módulo Tuner serán utilizados en el próximo acceso de la aplicación al punto del código modificado. Esta metodología puede ser solo aplicada a problemas que aparecen durante un número de veces en la ejecución de cada aplicación, aunque por otra parte suele ser habitual esta repetición dependiendo del paradigma de cómputo paralelo usado.

La herramienta MATE, y la sintonización dinámica que realiza, puede aplicarse a dos aproximaciones de sintonización:

1. Caja negra: No se proporciona conocimiento especifico de la aplicación. Esta

aproximación intenta poder sintonizar cualquier aplicación, y el desarrollador no debe prepararla para la sintonización (sin introducir cambios en el código fuente). En esta aproximación, lo relevante es intentar extraer información desde la aplicación desconocida. En este caso podemos centrarnos en los problemas relacionados con el paradigma usado para implementar la aplicación, y otras funcionalidades de bajo nivel comunes a las aplicaciones. Se puede entonces dinámicamente optimizar ciertas librerías que se usan para desarrollar la aplicación, en el caso de paso de mensajes, mediante PVM en nuestro caso de MATE, o bien optimizar usos comunes de la libreria C, como uso de memoria dinámica (optimizar el asignador o recolector de memoria). En el caso de PVM, pueden optimizarse las opciones de la libreria para adecuarlas mejor al entorno de ejecución [Mor03, Mor03a], en el caso de comunicaciones directas o indirectas, codificación (o no) de mensajes, etc.

2. Cooperativa: En este caso se asume que la aplicación es sintonizarle, y adaptable. El desarrollador, o bien ha adaptado su aplicación para los posibles cambios, mediante funcionalidades que se adapten a cambios en los puntos de sintonización, o bien el desarrollador utiliza un determinado framework de programación [Ces02, Ces99], que ha estado diseñado pensando en la sintonización. El desarrollador debe especificar el conocimiento que describe que tiene que medirse en la aplicación (puntos de medida), que modelo de prestaciones puede usarse para evaluar las prestaciones, y finalmente que hay que cambiar para obtener unas mejores prestaciones (puntos de sintonización). Algunas posibilidades en esta aproximación, teniendo en cuenta la existencia de información adecuada, son determinar y optimizar la carga de trabajo para aplicaciones Master Worker (sintonización de parámetros), adecuar el número de workers, seleccionar la estrategia más eficiente de distribución de datos (selección de implementación).

3.4. Conclusiones

El conocimiento profundo de las técnicas, y los elementos implicados en el análisis de prestaciones, es cada día más difícil de cubrir completamente. El apoyo al proceso de análisis que puedan dar las herramientas es imprescindible para superar esta brecha de conocimiento.

Las aproximaciones clásicas basadas en visualización no superan en su mayoría estos problemas, y obtienen un conocimiento más intuitivo que práctico sobre las causas de los problemas de prestaciones, o sobre las soluciones que hay que implementas.

Las herramientas automáticas van un paso más alla sistematizando las observaciones y deduciendo las causas que pueden llevarse al campo práctico del desarrollador de la aplicación. Existen, en las herramientas automáticas, diversas aproximaciones arquitecturales, dependiendo de su grado de automatismos, y de la pre-

cisión de la información requerida, generando un amplio espectro de herramientas para diferentes entornos con diferentes niveles de abstracción.

En la estrategia estática para herramientas automáticas se maximiza la información requerida para el análisis pero por contra se obtienen informaciones mucho más ricas en nivel de abstracción, que podrían llevarse directamente a la experiencia del desarrollador, en la implementación de sus aplicaciones. En el análisis automático estático con una aproximación de tipo post-morten, propondremos una arquitectura (ver en el siguiente capítulo) que nos permita los requerimientos iniciales que nos habíamos planteado en nuestro trabajo. En este tipo de arquitecturas estáticas genéricamente dispondremos de una serie de fases que se desarrollaran en cualquier herramienta con este modelo:

- Obtención de los datos iniciales, ya sea por diferentes técnicas de profiling (basadas en muestras, contadores o estados), o bien por trazas (en forma de flujos de eventos).
- Búsqueda e identificación de los problemas de prestaciones. Normalmente la herramienta dispondrá de algún tipo de clasificación jerárquica, o conjunto de problemas soportados, con una codificación interna de su búsqueda, o bien descripciones funcionales o estructurales del problema.
- Clasificación de los problemas encontrados, teniendo en cuenta las categorías existentes, y la importancia relativa de cada problema, respecto índices que maneje la herramienta para medir las ineficiencias causadas que se deriven en la aplicación.
- Selección (de todos o) de algunos de los problemas representativos, que compongan conjunto final de problemas de la aplicación.
- análisis de las causas de los problemas, teniendo en cuenta la instancia del problemas, sus datos parametricos recogidos, y su localización temporal y espacial en el código fuente, para determinar porque se ha producido. Esta etapa necesita acceso al código fuente de la aplicación.
- Selección de actuaciones a tomar para eliminar la causa, obtenidas del análisis del caso, en forma de sugerencia de actuación sobre el código de la aplicación.

En el siguiente capítulo desarrollaremos la arquitectura propuesta, así como el análisis y diseño de una prueba de concepto en forma de herramienta automática de tipo estático.

Capítulo 4

Propuesta de arquitectura para análisis automático de prestaciones

En este capítulo examinamos la funcionalidad de los diferentes componentes, de nuestra arquitectura propuesta, y su implementación en una prueba de concepto en la herramienta de análisis automático de prestaciones: KappaPI 2. Siguiendo un esquema lineal, se analizan las diferentes fases presentes en la arquitectura, teniendo en cuenta las ideas presentes y su implementación. Se describen los diferentes problemas de prestaciones analizados en los entornos de paso de mensajes, así como la información necesaria de análisis en forma de casos y condiciones que puede presentar el problema. Finalmente se comentan las diferentes técnicas empleadas para el análisis del código fuente, que nos permiten obtener la información final para exponer sugerencias sobre la mejora de la aplicación al usuario.

4.1. Conceptos, diseño y fases

Las aplicaciones que se ejecutan en entornos paralelos (y/o distribuidos) tienen que alcanzar ciertos índices de prestaciones, para conseguir plenamente los objetivos de los sistemas de cómputo de altas prestaciones. Por tanto el análisis de prestaciones comporta una tarea muy relevante, en la implementación de las aplicaciones paralelas.

Las librerías de paso de mensajes ofrecen al desarrollador un conjunto de primitivas (de comunicaciones, de sincronización) que no están disponibles en la programación secuencial. Desarrollar aplicaciones usando estas primitivas, y realizar la optimización pertinente de sus prestaciones son tareas complejas para usuarios no expertos, o con restringido grado de conocimiento de todos los componentes hardware y software que se ven implicados en este paradigma de programación.

Para poder ayudar al desarrollador en la sintonización, y en el análisis de prestaciones de su aplicación, se hace imprescindible disponer de herramientas de análisis de prestaciones. Pero, hay que indicar que existen muy pocas herramientas de prestaciones realmente útiles, y la aproximación, más popular (como vimos en los capítulos precedentes) era la realización del análisis de prestaciones a traves de herramientas de visualización [Cor95, Nag96, Ros98]. Estas herramientas nos proporcionaban una serie de vistas gráficas de lo que ocurría en la ejecución de nuestra aplicación [Sum00], que el desarrollador tenia que utilizar para realizar el proceso manual de análisis de prestaciones. Esta era una tarea difícil y gran consumidora de tiempo dentro del ciclo de desarrollo de la aplicación, y que requiere un alto grado de experiencia para identificar, superar los problemas de prestaciones, y alcanzar los índices de prestaciones esperados.

Para tratar estos problemas, y ayudar al usuario en la fase de análisis, se necesitaban herramientas ms potentes y amigables de cara al desarrollador. En este contexto, se desarrollan las herramientas automáticas de análisis de prestaciones (como vimos en el capítulo previo), como las ya mencionadas, Expert [Wol04], Scalea [Tru01], y KappaPI [Esp98].

Estas herramientas utilizan los ficheros de traza (combinados en ocasiones con medidas de profiling) tomados de la ejecución de la aplicación, e intentan detectar los problemas de prestaciones, mediante patrones de prestaciones (descritos bien en forma estructural, funcional o como conjunto de medidas). En la mayoría de estas herramientas, se utilizan aproximaciones con estrategias tipo automático post-mortem, que nos ofrecen la ventaja de ser capaces de considerar toda la información detallada obtenida durante la ejecución de la aplicación. Y además, nos permite que las etapas siguientes relacionadas con la posterior fase de análisis, no introduzcan ningún tipo de perturbación en la ejecución de la aplicación. Siendo únicamente el proceso de generación de traza de la aplicación, la que introduce perturbación en la ejecución, la cual puede minimizarse mediante diferentes técnicas (ver sección 2.7).

Las herramientas automáticas actuales, sufren algunas limitaciones, que tienen que ser superadas para disponer de herramientas plenamente útiles:

- En muchas de ellas, la especificación de los problemas de prestaciones, está codificada internamente en la herramienta, o bien tiene restricciones en las formas admisibles. Estas herramientas no pueden extenderse fácilmente para detectar, y/o analizar un conjunto grande de problemas. O bien especificar de una forma flexible nuevos problemas o métodos de análisis.

- Es bastante habitual en las herramientas, disponer de la capacidad de detectar los problemas, o su influencia en la aplicación, pero no suele ser habitual una identificación del punto preciso donde ocurren, relacionándolos con los puntos (o regiones) del código fuente de la aplicación.

- La información final proporcionada al usuario, no suele indicar que acciones deben ser llevadas a cabo para evitar/superar los problemas de prestaciones

que surgen en la aplicación.

La prueba de concepto, de la arquitectura presentada, KappaPI 2 es una herramienta de análisis automático de prestaciones, diseñada con la arquitectura propuesta, que es abierta para incorporar fácilmente el conocimiento de prestaciones de las aplicaciones paralelas, así como los problemas de prestaciones que aparecen en la disminución de la eficiencia de estas. La herramienta detecta y analiza los cuellos de botella de la aplicación debidos a la aparición de los problemas de prestaciones, y realiza sugerencias trasladables al usuario de como mejorar su aplicación.

En las ideas originales, de nuestra primera versión de la herramienta, KappaPI [Esp00], el objetivo del modelo de prestaciones estaba orientado a la reducción del tiempo total de ejecución, idea que se aproximaba por determinar los tiempos muertos (*idle*) de la aplicación. Se buscaban todos aquellas zonas, durante la ejecución, que suponían unos tiempos idle que superaban cierto umbral, como zonas donde había una caída de prestaciones, entonces se examinaban los eventos relacionados con la zona para determinar mediante una serie de reglas, que se iban aplicando mediante hipótesis de prestaciones iniciales, luego refinadas por los eventos observados, cual era la causa de la zona de tiempos idle.

En la nueva revisión de la arquitectura, y en particular en el producto en forma de la herramienta KappaPI 2, nuestro objetivo ha sido diseñar y implementar, una arquitectura para el análisis automático de prestaciones que cumpla los siguientes requerimientos:

- Especificación de conocimiento de prestaciones: Ofrecer mecanismos independientes de especificación que nos permitan introducir nuevo conocimiento, tanto de problemas de prestaciones, como del análisis de estos para determinar sus causas.

- La detección de los problemas de prestaciones debe ser guiada por la base de conocimiento (abierta) sobre problemas que se disponga. Los mecanismos de detección son independientes de los problemas concretos que se dispongan en cada momento.

- Independencia del sistema de paso de mensajes utilizado. La herramienta debe trabajar con modelos abstractos de los diferentes elementos presentes en un entorno de paso de mensajes. Siendo en particular independiente, de los formatos de ficheros de trazas utilizados, así como de las primitivas de paso de mensajes utilizadas.

- Relacionar los problemas con el código fuente de la aplicación: Debe ser capaz de especificar precisamente que puntos de código se relacionan con un determinado problema (¿donde aparece?), examinar posibilidades de alteración de ese código (¿que podemos realizar?), y ofrecer soluciones a sus causas (¿Porqué apareció?).

La arquitectura propuesta, y su ejemplo en la herramienta KappaPI 2 (que iremos desarrollando a lo largo de este capítulo) la podemos resumir en la figura 4.1

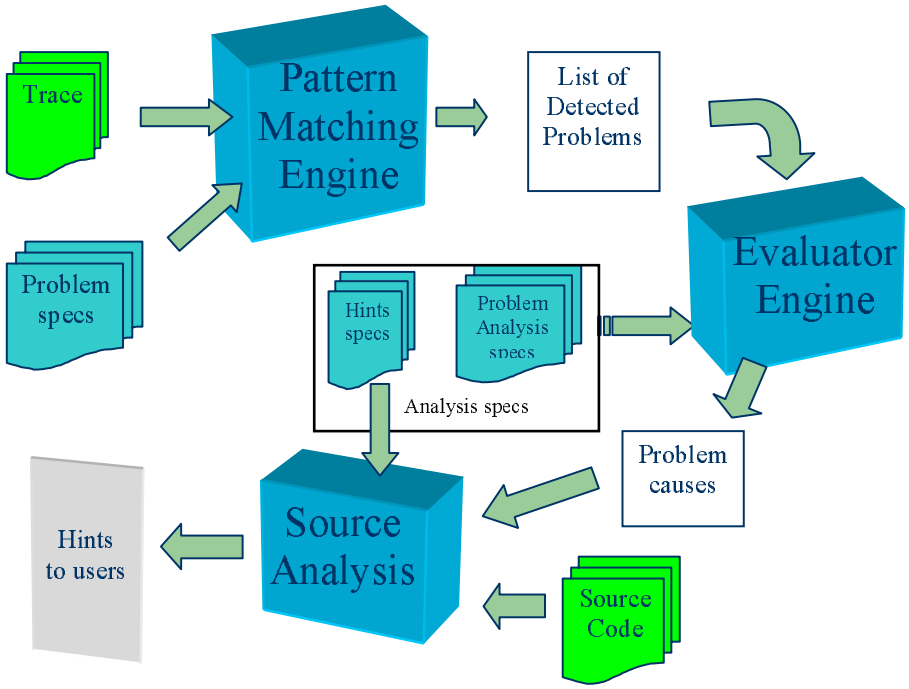

Figura 4.1: Arquitectura de la herramienta automática de prestaciones KappaPI 2

En KappaPI 2, el primer paso es la ejecución bajo control de una herramienta de monitorización basada en traza (independiente de KappaPI 2) para el entorno de paso de mensajes utilizado (ya sea MPI o PVM) que captura todos los eventos (instrumentados) relacionados con las primitivas de paso de mensajes que ocurren durante la ejecución de la aplicación. La herramienta utiliza los ficheros de traza, y la base de conocimiento sobre los problemas de prestaciones, como entradas para la detección de patrones de problemas de prestaciones definidos desde un punto de vista estructural. Después, se clasifican los problemas encontrados dependiendo de determinados índices. Y para los problemas seleccionados, se realiza un análisis de causas, utilizando inspección y análisis del código fuente de la aplicación, para finalmente proporcionar un conjunto de recomendaciones al usuario, indicando como debe modificar el código fuente de la aplicación para evitar los problemas detectados.

Mencionamos a continuación, ampliando algunas similitudes y las diferencias principales, los trabajos existentes en el campo de las herramientas automáticas, que más directamente están relacionadas con la arquitectura propuesta, y las similitudes y diferencias de otras herramientas con el prototipo de la herramienta propuesta KappaPI 2.

4.2. Trabajos relacionados

En la sección 3.3 comentamos algunas herramientas de análisis automático presentes en la literatura, comentamos aquí algunas de las ideas que nos sirvieron de inspiración para nuestro trabajo, y otras que ofrecen visiones complementarias, u otras aproximaciones, así como las principales diferencias/aportaciones de nuestra herramienta KappaPI 2 respecto a las existentes.

En el caso de Expert [Wol00, Wol04], quizás la herramienta con una concepción más similar a la nuestra, la herramienta está pensada para entornos SMP en cluster (entornos híbridos SMP), para entornos de programación MPI, OpenMP y la combinación de ambos. En la herramienta, se definió el concepto de evento compuesto (ya utilizado tambien en [Fox98]), pero su especificación no se hace por un lenguaje independiente sino que se utilizan lenguajes de tipo script, para realizar un ”programaçompleto que trate el proceso de detectar un problema concreto, esta descripción incluye una estructura tipo evento compuesto, más la codificación en el lenguaje script escogido (tcl, python) de las condiciones e instancias que tienen que cumplir los eventos, mediante una API denominada EPILOG, que es la que interacciona con el/los ficheros traza de la aplicación. Y se implementan también los cálculos a realizar con los datos de los eventos. El análisis determina que problemas han aparecido, y los porcentajes que cada problema implica en el rendimiento global o parcial (en función o bloque de código) de la aplicación, estableciendo (según el análisis multidimensional que realiza) un ranking de problemas, llamadas (teniendo en cuenta el call graph dinámico), y localización en el código. Dispone de una herramienta denominada CUBE, que realiza la visualización de estos datos, y es capaz de visualizar las diferencias entre los datos recopilados de varios experimentos con la misma aplicación. Expert no realiza ningún tipo de sugerencia respecto al código de la aplicación de cara al usuario final. Algunas de las diferencias principales de KappaPI 2 respecto a Expert son:

- a) La especificación de los problemas se realiza desde un punto de vista estructural de eventos que componen el problema, por contra Expert utiliza programación funcional (en lenguajes script o orientados a objeto).

- b) La especificación en Expert se basa en su API de traza (EPILOG), que se necesita conocer para definir funcionalmente el problema, en el caso de KappaPI 2, el uso de la API es interno, solo se necesita conocer los tipos de eventos y sus características.

- c) Además KappaPI 2 ofrece mecanismos de abstracción de trazas de aplicaciones de paso de mensajes, podemos usar diferentes trazadores para los entornos de PVM y MPI, y realizar conversiones a las trazas internas.

- d) Expert no ofrece técnicas para el análisis del código fuente, o recomendaciones de cara al usuario final para mejorar su aplicación.

- e) En KappaPI 2 se añade un segundo nivel de especificación para el análisis de las causas que han provocado el problema. Se puede introducir en la herramienta conocimiento sobre los problemas y su proceso de análisis.

En Scalea [Fah00], y en particular en JavaPSL [Ser03], se utiliza la orientación a objetos, como mecanismo para construir el árbol (o jerarquía) de las propiedades de prestaciones, mediante la especialización y composición a partir de algunas iniciales existentes. En este caso se utilizan mecanismos propios, del lenguaje Java, para la especificación. KappaPI 2 hemos realizado una especificación solo estructural independiente del lenguaje, definiendo toda la estructura de los problemas en datos XML, dejando fuera el cómputo y las estructuras relacionadas, y en particular la jerarquía de problemas, no existe como tal inicialmente, sino que se crea en ejecución (con posibilidad de guardarse previamente), lo que denominamos el árbol de decisión, estructura jerárquica de los problemas, según el orden y aparición de los eventos que componen los diferentes problemas.

En Paradyn [Mil95], la utilización de métricas basadas en contadores guia la evolución de la búsqueda de problemas, en contraste se utilizan en KappaPI 2 descripciones de los problemas de prestaciones en forma de estructura de eventos presentes en el problema y sus relaciones. Así como se introduce también la idea, en KappaPI 2, de abrir la especificación del análisis de las posibles causas del problema. Paradyn utiliza monitorización de métricas on-line, mientras en nuestro caso utilizamos trazas con análisis post-mortem.

4.3. Eventos

En los sistemas basados en eventos, la caracterización de evento intenta definir el suceso de una acción atómica en una localización definida, en un instante particular del tiempo.

En nuestro caso, cuando modelamos una aplicación paralela, en el entorno de paso de mensajes, cada tarea de la aplicación realiza una secuencia de acciones a lo largo de su ejecución, en las diversas areas de cómputo, comunicación y sincronización.

En la utilización de entornos de paso de mensajes, las operaciones primitivas se hacen con llamadas a funciones de librería (de la API que se este utilizando), mediante el uso explicito de paralelismo. Los eventos [Kun93, Kun94] que definamos van relacionados intrínsecamente con las operaciones de paso de mensajes utilizadas, y en muchos casos no pueden definirse como atómicas. Ya que una comunicación, ya sea envío-recepción, o bien una operación colectiva, estaría compuesta de una serie de sucesos normalmente pertenecientes a diferentes tareas, y sucediendo en diferentes instantes de tiempo.

En los entornos de paso de mensajes, disponemos pues, de diferentes operaciones explicitas de comunicación: como comunicaciones punto a punto, colectivas, y sincronización. En este caso, necesitamos de una definición de evento que nos per-

mita establecer como suceso atómico, la unidad elemental que podemos observar, y a partir de la cual podremos luego componer las operaciones de paso de mensajes.

En nuestro caso, partimos de la observación recogida mediante monitorización, en la traza de la ejecución de la aplicación. En esta hemos utilizado mecanismos que nos permiten capturar las llamadas producidas de forma explicita a primitivas de paso de mensajes. Y generalmente, como veremos, en el siguiente sección, los sistemas de traza capturan las primitivas, en función del punto de llamada, en cuanto a localización de tarea, y tiempo en que se produjo.

Cada operación que se produzca, en una determinada tarea, en la aplicación sera descrita por unos tiempos donde se producirá el evento de inicio y final de la llamada de la primitiva. Así en una tarea particular, el establecimiento o participación en una comunicación (o sincronización) se observara como una serie de eventos de principio y final de un tipo de operación.

Así el sistema de traza de eventos que se use, puede ver la ejecución de una aplicación como una secuencia de acciones que se han producido, y que se consideren relevantes para los propósitos de la observación y posterior análisis del comportamiento de la aplicación. En el caso particular del análisis de aplicaciones por paso de mensajes, sus llamadas a primitivas del entorno, y los tiempos de entrada y salida.

Con esto tendremos que seleccionar una serie de tipos de eventos que podamos/queramos observar, y así nos definirán la vista observada (y/o modelo), que obtendremos desde la traza sobre la aplicación ejecutada.

Para esto, tendremos que definir un modelo de eventos, en el que incluiremos, los tipos de eventos, junto con una serie de atributos y restricciones asociados a cada tipo de evento.

4.4. Generación de trazas

Una de las primeras fases en cualquier entorno de prestaciones es la obtención inicial de la información de lo que ha sucedido en la aplicación paralela. Para esto utilizamos en nuestro entorno, ficheros de traza obtenidos desde la ejecución de la aplicación.

Dado el soporte de de varios entornos de paso de mensajes (PVM y MPI en nuestro prototipo), se hace necesario disponer de varias herramientas de generación de traza para cada entorno. Los requerimientos básicos consisten:

- Nos tienen que permitir capturar los eventos de llamadas de primitivas de paso de mensajes, midiendo los tiempos de duración de estas, por ejemplo mediante eventos de entrada y salida de una determinada primitiva.

- Tenemos que tener la información parametrica de cada primitiva, sus parámetros de llamada, y otra información que puede deducirse fácilmente de ellos, de forma dinámica (tamaños de mensajes, de grupos de procesos, relaciones

de los eventos, etc.). Esta información compondrá los atributos de cada tipo de evento.

- Unos parámetros, para nosotros, imprescindibles es la información referente al código fuente, tenemos que soportar información referente a ficheros, y números de linea donde se encuentren las primitivas a las que se halla producido llamada. O sea, cada evento a demas de información parametrica, y temporal, deberá poseer información espacial de su posición dentro del código fuente de la aplicación.

Observamos a continuación, que tipo de información, y por tanto modelo de eventos, puede proporcionarnos cada una de las herramientas de monitorización usadas (en PVM y MPI), y como hemos generalizado estas herramientas, a un modelo abstracto de eventos para el uso interno de la herramienta, que nos abstrae del modelo particular de cada entorno.

4.4.1. Obtención: herramientas TapePVM y MPITracer

En el entorno PVM se ha utilizado la herramienta de traza TapePVM [Mai95], desarrollada en los laboratorios LMC-IMAG, para la monitorización de programas paralelos basados en la la librería PVM. Ofrece un conjunto de funciones en C que genera información de traza en un formato propio, trasladable a otros como PICL. La herramienta está diseñada tomando en cuenta determinadas técnicas para disminuir la perturbación en la aplicación (buffering local a la tarea de la traza de esta), y usa temporización global sincronizada para todas las tareas de la aplicación.

Esta herramienta de traza se basa en el concepto de *wrappers* de primitivas PVM. El código PVM inicial de la aplicación es convertido por unas herramientas que instrumentan cada primitiva, de manera que se ve substituida la llamada API inicial, por una nueva llamada de la librería de TapePVM.

Esta nueva llamada, se encarga de registrar los puntos de entrada y salida de la primitiva original de PVM, así como de captar la información dinámica que se ha aportado y/o obtenido en la llamada, así como dependiendo de la llamada, operaciones adicionales de capturar (o generar) información extra referente a los eventos, así como de la localización física (en código fuente) de las llamadas producidas. En la siguiente tabla mostramos los campos básicos que permite obtener para cada evento:

El trazador TapePVM trabaja en varias fases:

1. Inserción en el código fuente, una llamada de inicialización de TapePVM, para que capture los eventos de la librería PVM, y genere una nueva versión de código fuente con esta información incluida, juntamente con los wrappers a las primitivas PVM

2. El nuevo código se compila y enlaza de nuevo (manualmente).

Atributo	Notas
alpha	retardo en micro-segundos para generar el evento
type	tipo de evento (primitiva o operación usada
task	Identificador de la tarea donde se ha producido
file	Id del fichero fuente
line	Linea del fichero fuente conteniendo la llamada pvm
dates	Tiempo global en segundos
dateus	Tiempo en micro-segundos
varpart	Atributos adicionales relacionados con el tipo de evento

Tabla 4.1: Atributos de los eventos en TapePVM

3. Cuando el código PVM se comienza a ejecutar, se realiza una fase de sincronización de tiempo para averiguar las diferencias de tiempo entre las máquinas, y establecer el tiempo global a todo el sistema.

4. Acabada la fase de sincronización, el programa continúa con la ejecución normal de la aplicación, donde se recogerán en buffers, los eventos (junto con sus datos) que se vayan produciendo, y según la configuración del trazador se volcaran periódicamente a disco.

5. Cuando todos las tareas PVM acaban, TapePVM junta en un solo fichero de traza, toda la información obtenida sobre todos los que eventos que han generado cada tarea individualmente y transforma los tiempos locales en tiempo global.

Una desventaja que podemos indicar referente a este tipo de herramientas de traza, con técnicas de *wrapper* de primitivas, es que cada vez que tengamos que realizar cambios en la aplicación original, hemos de generar de nuevo la copia con la monitorización insertada, compilarla, y probar de nuevo su nueva ejecución, haciendo este proceso bastante tedioso.

Además la herramienta en si no es flexible, en el sentido que no es fácil adaptar o incorporar nuevos eventos, o atributos a los eventos existentes que podrían ser útiles para el análisis posterior. Aunque es plausible realizar algunas de estas incorporaciones, debido a la disposición del código fuente de la herramienta; se necesitaría un trabajo de adaptación de la herramienta para cada tipo de monitorización que se quiera realizar para aplicaciones diferentes. Por contra, si se soportan ciertos filtrados de eventos, o monitorización selectiva, al poder habilitar o deshabilitar la monitorización de tipos de eventos, pudiéndose actuar sobre eventos de comunicaciones punto a punto, colectivas, o otros tipos. Pudiéndose así filtrar solo los eventos que interesasen de la aplicación.

Respecto a la alternativa de monitorización que se ha usado en entornos MPI (de tipo MPICH), nos encontramos con la herramienta MPITracer [Iva04]. MPITrace es una herramienta de traza, basada en instrumentación dinámica, para aplicaciones MPI en entorno MPICH (una de las implementaciones de dominio publico

de la especificación MPI).

Esta herramienta, desarrollada en nuestro grupo, se encontraba en un estado de prototipo, y hemos aumentado su funcionalidad para cubrir los requerimientos de monitorización iniciales necesarios para nuestra herramienta de análisis, y las ampliaciones necesarias para capturar los diferentes eventos de las llamadas a primitivas MPI.

Entre las características finales de la nueva versión desarrollada de MPITracer podemos destacar:

- Monitoriza las llamadas primitivas a MPI, en el entorno MPICH para dispositivo ch_p4. Pudiendo monitorizar llamadas de tipo p2p, colectivas y de sincronización.

- Los puntos (primitivas) a monitorizan se pueden especificar por cada aplicación en ficheros de configuración del trazador. Cada aplicación dispondrá de su lista de puntos a monitorizar. Pudiendo monitorizar en aquellos puntos que nos interesen, una serie de primitivas concretas, o todas las que queramos. De esta manera el usuario informa al trazador, a traves de un fichero de entrada, de los puntos del programa sobre los cuáles quiere obtener información y del tipo de información que quiere obtener. También disponemos de ficheros, de puntos de instrumentación, generales para monitorizan conjuntos predeterminados de primitivas, que pueden extenderse con ficheros personalizados para tipos de eventos relacionados con la aplicación, o incluso definir puntos de instrumentación para funciones de usuario en la aplicación.

- No es necesario alterar el código fuente de las aplicaciones a monitorizar, ya que se utiliza instrumentación dinámica, mediante el entorno DyninstAPI [Buc00, Hol02]. Esta API de instrumentación dinámica nos permite poder colocar los puntos de monitorización indicados anteriormente en la aplicación de forma dinámica durante la ejecución de la aplicación.

- La aplicación en código binario (ni en formato fuente) no necesita ningún cambio, a la hora de aplicar el trazador. Únicamente es necesaria la inclusión de información de depuración en los ejecutables de la aplicación (requisito de DyninstAPI).

MPITracer nos permite así indicar una serie de puntos del programa a monitorizar, indicando puntos de entrada y salida de las funciones a monitorizar, y obtener los puntos del código (localización espacial en el código fuente) donde se realiza una llamada a función monitorizada. De estos puntos podemos obtener información sobre:

- Máquina (Host monitorizado) donde se ejecuta el proceso.

- Fichero fuente y linea donde se hace la llamada a la función monitorizada.

- Tiempo de entrada y salida global (a la aplicación) de las funciones monitorizadas.
- En el caso de MPI, comunicador global y identificador de proceso MPI dentro del comunicador, y los parámetros asociados a las comunicaciones punto a punto, o colectivas dependiendo del caso.

En el caso de MPITrace los atributos que pueden obtenerse de un evento estándar son los siguientes:

Atributo	Notas
stamp	Ocurrencia temporal respecto tiempo global de la aplicación (en segundos)
stamp_us	tiempo en micro-segundos añadido al anterior
host	Máquina o nodo donde ocurrió el evento
comm	Comunicador (MPI) usado y rango de la tarea dentro de este
type	tipo de evento (primitiva o operación usada) según especificación del usuario en el trazador
Entry	Evento entrada o salida de la región de la primitiva
line	Linea del fichero fuente
file	Id del fichero fuente
varpart	Atributos adicionales relacionados con el tipo de primitiva del evento

Tabla 4.2: Atributos de los eventos en MPITracer

En esta herramienta de monitoración, al basarse en una aproximación dinámica, podemos cambiar fácilmente la información que recogemos, mediante los ficheros de configuración de MPITracer, ya sea globales, o personalizados para una aplicación dada. En particular mediante la parte variable de los eventos, podemos obtener información adicional en forma de atributos para los eventos, mediante la indicación de que primitivas pensamos capturar, y cuáles de sus parámetros serán capturados, o por otra parte introducir cálculos adicionales de forma dinámica, que nos permitan recoger la información que nos haga falta para estimar un atributo adicional.

MPITracer, permite una flexibilidad relevante del sistema de monitorización, de manera que podemos adaptar fácilmente la herramienta de traza para que nos proporcione los datos que realmente nos interesan de cada una de las primitivas.

4.4.2. Abstracción PVM y MPI

Uno de los conceptos significativos en la fase de obtención de la traza, en la herramienta KappaPI 2, es la abstracción interna de la traza usada por la herramienta. El objetivo es poseer una abstracción de los eventos que se pueden dar en un paradigma de paso de mensajes [Moh93], de forma que tanto las primitivas de

paso de mensajes de los entornos como PVM o MPI, puedan adaptarse a la traza interna. Así como otros entornos de paso de mensajes que pudieran utilizarse.

Para la creación de esta especificación interna de traza, se han tenido en cuenta los siguientes requerimientos, en cuanto al modelo de eventos necesario:

- Soportar una abstracción genérica de las operaciones típicas presentes en cualquier entorno de paso de mensajes. En este caso se han proporcionado tipos de eventos, que crean abstracciones de eventos comunes presentes en entornos PVM y MPI. Incluyendo operaciones punto a punto, diferentes operaciones colectivas (scatter, gather, reduce), y operaciones de sincronización (típicamente barriers).

- Teníamos que tener disponible la información de las tareas presentes en la aplicación, y posiblemente de las que pudieran aparecer dinámicamente. Se proporciona soporte así a eventos de inició, finalización de tareas, y en particular al inició dinámico de estas (por ejemplo mediante operaciones de spawn).

- Soporte para operaciones especializadas. Aún teniendo en cuenta el requerimiento de abstracción de paso de mensajes, hay que tener en cuenta que en los sistemas de paso de mensajes, es habitual la inclusión de primitivas especializadas para ciertos tipos de comunicación. En estos casos, cuando las primitivas son de amplio uso, es posible incorporarlas a la abstracción, aunque solo formen parte de alguno de los entornos de paso de mensajes. Por ejemplo podemos señalar el soporte, de operaciones mcast en PVM, o soporte para operaciones de tipo All en MPI.

- Los atributos especificados en los eventos, tienen que ser lo suficientemente genéricos, para que estos se puedan obtener fácilmente, o puedan calcular a partir de información existente en los entornos. Cualquier información extra, que pueda incluirse en los eventos, debe poderse obtener en cualquiera de los sistemas de monitoración, y que esta información sea de obtención directa, o con simple computación, para no aumentar la perturbación de las herramientas de traza. Se intenta solo obtener información directa desde las llamadas de primitivas, o resolverla con llamadas adicionales lo más simples posibles.

- En particular, en el caso de los atributos, es imprescindible la incorporación de la información del código fuente, de manera que cada evento tenga una relación directa con la posición del código que lo ha producido (linea y fichero fuente).

Con estos requerimientos, hemos definido un modelo simple de eventos abstractos para el paso de mensajes, que utilizamos internamente en la herramienta, y que nos permite modelar los sucesos de gran parte de las primitivas de PVM y MPI, a partir de la información obtenida por las herramientas de monitorización

por traza como TapePVM y MPITrace. Así como la fácil conversión, que puede realizarse, desde otras posibles herramientas de monitorización basadas en traza para los entornos PVM y MPI.

Atributo	Notas
Stamp	Tiempo (en sec y micro-sec) global de ocurrencia
task	Identificador de la tarea donde se ha producido
type	tipo de evento (primitiva o operación usada
file	Id del fichero fuente
line	Linea del fichero fuente
Entry	Entrada o salida de primitiva
varpart	Atributos adicionales relacionados con el tipo de evento

Tabla 4.3: Atributos generales en la abstracción de traza

En particular respecto a los tipos de eventos existentes hay que observar las diferentes categorías relacionadas con las primitivas de paso de mensajes. Pudiendo separar los tipos en:

- Punto a Punto: Eventos de comunicación relacionados con primitivas de comunicaciones punto a punto. En particular relacionadas con las operaciones de envío (send), recepción (recv) y los diversos modos que pueden presentarse en estas comunicaciones.

- Punto a Punto compuestas: Eventos relacionados con primitivas compuestas. En algunas librerías, se optimizan algunas comunicaciones punto a punto agrupándolas en una sola primitiva (por ejemplo caso de MPI_sendrecv).

- Colectiva para todos: Eventos relacionados con primitivas donde todos los participantes llaman a las mismas primitivas. Pudiendo tener comportamientos iguales en todas las tareas, o bien diferir respecto una tarea que actúe con el rol de tarea root.

- Sincronización: Eventos relacionados con primitivas de sincronización. En particular estos pueden considerarse un subconjunto del anterior tipo, como colectivas sin tarea especializada en el rol de root.

- Colectiva con punto a punto: En algunos casos, se producen llamadas que producen una comunicación colectiva, a partir de una tarea con rol de root, pero el resto de las tareas participantes utilizan primitivas punto a punto (caso por ejemplo de pvm_mcast).

- Otros: Eventos relacionados con inició, finalización o configuración de comunicaciones. Eventos como *Enroll*, para inicio de la tarea; *Exit* para la finalización, o otros referentes a tratamientos de datos, como en/desempaquetar.

4.5. Especificación de problemas

El análisis de prestaciones es un campo en constante evolución, que tiene que tener en cuenta los diferentes factores asociados a los entornos de programación, sistemas operativos, y las facilidades existentes de hardware y red que sean utilizadas. No es un conjunto de conocimiento cerrado, y por tanto hay que adaptarse a las nuevas situaciones que puedan aparecer. Es importante, que las herramientas automáticas de análisis sean diseñadas de forma abierta y ampliable, de manera que el conocimiento relacionado con nuevos problemas de prestaciones pueda ser incorporado de forma fácil a la base de conocimiento de la herramienta. Se requiere una forma de especificación abierta para poder añadir nuevos problemas. En este contexto es necesario desarrollar elementos que nos permitan la definición de nuevos problemas de prestaciones. ASL (APART specification Language) es un buen ejemplo de esta aproximación [Fah00, Fah01a, Ger02]. En nuestro caso, como veremos, hemos utilizado una aproximación por definición estructural de los problemas, definiendo los eventos relacionados, las restricciones en el tiempo y localización, y algunos cálculos en forma de índices para evaluar posteriormente la importancia relativa del problema de prestaciones.

4.5.1. Eventos y patterns en sistemas de paso por mensajes

Un evento consiste en un cambio del estado del sistema. En nuestro caso, en un sistema de paso de mensajes, relacionamos un evento, con la ocurrencia de una determinada primitiva de paso de mensajes, y otros estados especiales como, por ejemplo, los de inicio y final de una tarea (ya sea de forma estática o dinámica).

En la ocurrencia de una primitiva, nos encontramos con dos eventos dentro de una tarea analizada: 1) El inicio de la operación, que nos provoca el cambio de estado de la tarea, para iniciar los mecanismos que le permitan comenzar la operación. Y 2) la finalización de la operación, cuando esta se da por finalizada, y puede recuperarse el estado previo en que se encontraba la tarea, o bien iniciarse la operación siguiente.

Los eventos que describamos del sistema de paso de mensajes, tendrán unos campos básicos (atributos) como:

- TimeStamp: O marca de tiempo global al sistema, teniendo en cuenta que los tiempos locales de las tareas hayan estado sincronizadas en un tiempo global.
- Localización: El evento se ha producido en un determinado lugar del sistema paralelo (o distribuido), en un determinado nodo (si se dispone de varios procesadores), procesador, y proceso (normalmente representado por el identificador de la tarea).
- Campos adicionales: Según el tipo de llamadas primitivas usadas, y la información que sea posible recolectar. U otros atributos referidos a correspondencias entre eventos (p.e. relaciones envío - recepción).

A esta información típica, en nuestro caso, debemos añadir una información de localización extra, respecto al código fuente, en forma de linea, y fichero, donde se encuentra la llamada producida. Para poder implementar los requerimientos de información sobre el código fuente de la aplicación.

De esta forma la traza de nuestras aplicaciones, dentro de un sistema de paso de mensajes, la podemos ver como una secuencia ordenada de eventos respecto a sus marcas de tiempo (global al sistema).

También podemos definir el concepto de estado, por el sumario de actividades de un determinado instante temporal, representado por el conjunto de sucesos que se encuentran en un estado de iniciado (han superado su evento de inicio pero no el de final).

Para el concepto de patrón de eventos, relacionaremos un conjunto de eventos, que dispondrán de una localización, así como de la relación de pertenencia a una o más operaciones de paso de mensajes en curso. Para expresar estas relaciones usaremos el concepto de eventos compuestos (formalizado en [Wol00])

4.5.2. Apart Specification Language (ASL)

Una nueva aproximación a la formalización de la especificación del conocimiento en prestaciones, y los datos asociados, se ha desarrollado en el APART Specification Language (ASL) [Fah00, Fah01a], que se desarrollo dentro del grupo de trabajo APART en el proyecto *Automatic Performance Analysis: Resources and Tools* [APA99]. ASL proporciona un lenguaje formal para describir las propiedades de prestaciones relacionadas con diferentes modelos de programación. Un propiedad de prestaciones representa un aspecto del comportamiento, que tiene que ser evaluada con los datos de comportamiento de la aplicación para verificar una serie de condiciones y limites, para determinar si finalmente crea un problema de prestaciones (es decir, una determinada propiedad cumpliendo unas condiciones, y que sobrepase los limites para tenerla en cuenta).

Este lenguaje ASL, es útil para el objetivo de eliminar las dependencias codificadas internamente en las herramientas de los problemas específicos a tratar. Conseguimos sacar el código de implementación de los problemas, a un lenguaje de especificación externo.

ASL como tal, en nuestra aproximación de herramienta, no nos era especialmente útil, ya que en su mayor parte se encuentra basado en conceptos de profiling, donde las propiedades son evaluadas por métricas obtenidas por estos métodos. Otra aproximación [Wol00], incluyo una extensión ASL para incluir el concepto de evento compuesto, en base a usar instancias de eventos junto con sus relaciones, para encontrar propiedades de prestaciones basadas en eventos (en lugar de métricas de profiling) en los sistemas basados en traza.

4.5.3. Especificación en KappaPI 2

En KappaPI 2 uno de los objetivos es abstraer la especificación de problemas de prestaciones. Al tratarse de una herramienta basada en traza, se hizo natural el concepto de evento compuesto para definir los diferentes problemas de prestaciones, que ya formaban parte de la herramienta inicial KappaPI, y de otros desarrollados o adquiridos de la literatura de análisis de prestaciones. El lenguaje ASL, o más bien, las extensiones para eventos compuestos, se presentaban como candidato natural para la especificación.

Pero se presentaban algunos problemas en particular: a) La búsqueda de interoperabilidad entre diferentes herramientas dificultaba el uso de ASL como tal, no existían ni analizadores léxicos ni semánticos para el lenguaje, construir alguno de particular para una plataforma concreta no hacia viable la interoperabilidad con otras herramientas, sino había un conjunto de analizadores disponibles para varias plataformas, y más o menos integrados con diferentes herramientas. b) La verificación de relevancia de la propiedad, basada en limites, puede depender fuertemente del modelo de prestaciones en que se base la herramienta, y depende de como se gestione puede crear incoherencias entre las diferentes propiedades. En este caso era mejor aislar el método para clasificar los problemas, basándose en el comportamiento que presentaran durante la ejecución, en localización, tiempo, y repeticiones que presentaran. c) Respecto b) una de las ideas es que precisamente esta ïmportanciaëra lo que diferenciaba el concepto de propiedad, respecto del de problema de prestaciones. Una aproximación alternativa era suponer una pura definición estructural del problema de prestaciones, y seria posteriormente el modelo de clasificación de estos el que aceptaría o (descartaría) el problema como importante, y finalmente la etapa de búsqueda de causas y sugerencias, la que determinara si podía dar alguna solución al problema planteado, o por si contra se trata de un problema intrínseco a la aplicación, y es necesario subir de nivel de abstracción para atacar la problemática que causa la disminución de prestaciones.

En este caso se decidió, implementar una definición estructural del problema, consistente en: Eventos presentes en el problema, condiciones de relación entre los eventos (que tipos, quien estaba relacionado con quien, número de instancias presentes, etc.), y el calculo de métricas necesarias para su posterior clasificación. El lenguaje escogido, fue una implementación libre (teniendo en cuenta las consideraciones del apartado anterior) de ASL en XML, por su independencia de la plataforma utilizada, y la facilidad que incorporaba para conectarse con otras herramientas, y la amplia disponibilidad de APIs para el análisis tanto léxico como semántico de XML.

Entonces la especificación usada en KappaPI 2, puede verse como una traslación simplificada en XML de las propuestas de ASL, utilizando el concepto de eventos compuestos [Wol04], para describir los problemas de prestaciones.

La especificación, está basada en la estructura de eventos presentes en el problema de prestaciones (ver cuadro), con un evento inicial, denominado ROOT, que es el primer evento que detectamos como perteneciente al problema en la primera

tarea analizada, y que desencadena el resto del problema. Seguido por diversas instancias de eventos (sección INSTANTIATION), y varias restricciones temporales, y otras relacionadas con los eventos de las tareas participantes.

En el siguiente ejemplo, vemos una parte de esta especificación, correspondiente a un problema de prestaciones relacionado con un emisor bloqueado (Blocked sender), ver figura 4.2 para la representación gráfica.

En este problema, una tarea está esperando en una operación de recepción de un mensaje, debido a que la tarea que lo emite está a su vez bloqueada por otra operación de comunicación previamente iniciada.

Especificación Blocked Sender

```
1  <PATTERN Name="Blocked Sender">
2      <ROOTTYPE>RECV</ROOTTYPE>
3      <INSTANCES>
4          <EVENT NAME="S1" TYPE="SEND" TO="ROOT"></EVENT>
5          <EVENT NAME="S2" TYPE="SEND" TO="R2"></EVENT>
6          <EVENT NAME="R2" TYPE="RECV" FROM="S2"></EVENT>
7          ...
8      </INSTANCES>
9      <CONSTRAINT>
10         ...
11         <COND TYPE=">" OP1="E2.stamp" OP2="E1.stamp">
12         <COND TYPE=">" OP1="E2.stamp" OP2="E3.stamp">
13         <COND TYPE="=" OP1="E3.taskId" OP2="E2.taskId">
14     </CONSTRAINT>
15     <EXPORT>
16         <COMPUTE NAME="idle_time" AS="-"
17         OP1="E2.stamp" OP2="E1.stamp"></COMPUTE>
18     </EXPORT>
```

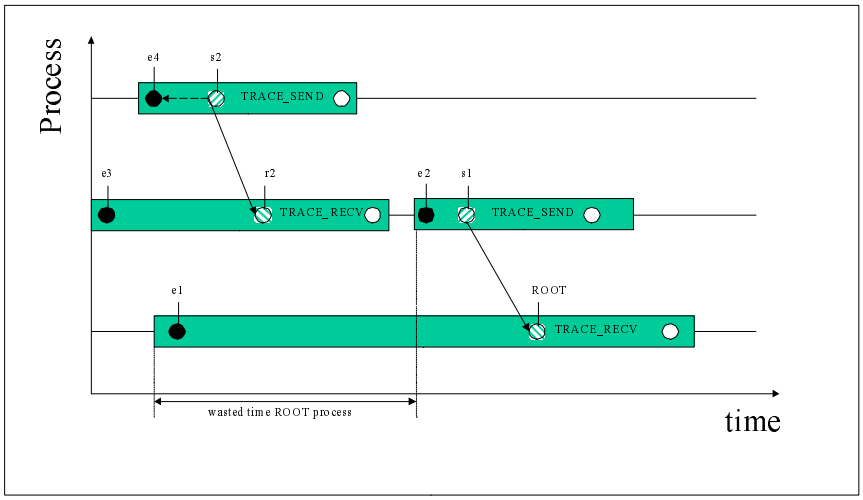

Figura 4.2: Gráfico de la estructura de un problema de emisor bloqueado

Tres tareas están participando en este problema, y cuatro eventos (de inicio) están implicados. La tercera tarea tiene un tiempo perdido debido al receptor bloqueado por una operación de envío aún no iniciada en la segunda tarea. Este envío no se ha iniciado debido a que la tarea está bloqueada en una operación de recepción respecto la primera tarea.

Esta especificación estructural de las ineficiencias de las aplicaciones por paso de mensaje, nos permite que después, en la fase de detección de la herramienta, podamos buscar las ineficiencias de la aplicación a partir de nuestra base de conocimiento, por medio de correspondencias simbólicas entre las estructuras definidas en la especificación respecto los datos de la ejecución de la aplicación en forma de eventos del sistema de paso de mensajes.

Esta especificación de los problemas evitan el uso de conocimiento internamente codificado en la herramienta (como era el caso de la primera versión de KappaPI), que se hacia posteriormente poco manejable, y escalable. Y por tanto nos permite mantener la herramienta KappaPI 2, abierta a la incorporación de cualquier nuevo conocimiento que surja sobre nuevos problemas de prestaciones, o redefinición de los existentes.

En particular, un punto interesante, concierne a la personalización de la herramienta, dependiendo de la base de conocimiento que habilitemos en la herramienta, tenemos una instancia diferente de KappaPI 2. Un conjunto de problemas especificados define una instancia de KappaPI 2, y en este sentido la arquitectura flexible mostrada en el trabajo, permite utilizar KappaPI 2 como un laboratorio para experimentar diferentes definiciones de problemas, y diferentes conjuntos de problemas con que analizar las prestaciones de una aplicación. En este sentido hablamos de KappaPI 2 como un ejemplo de la arquitectura, y no una mera herramienta concreta de prestaciones, la personalización define la herramienta de prestaciones.

En nuestro trabajo, hemos personalizado KappaPI 2 con un catalogo de problemas, que nos define un rango significativo de las ineficiencias clásicas aparecidas en las aplicaciones de paso de mensajes. Estos problemas, provienen de problemas clásicos conocidos del modelo de paso de mensajes, de nuestros trabajos previos en la herramienta KappaPI, la experiencia adquirida en el desarrollo de aplicaciones paralelas [Jor98, Jor01] y de otros problemas presentes en la literatura de prestaciones. De los cuáles hemos producido las especificaciones para la herramienta KappaPI 2, y hemos analizado los casos de uso que se observan en su aparición en las aplicaciones, introduciendo en la herramienta la especificación de su análisis de causas (que veremos en las siguientes secciones).

4.6. Catalogo de problemas en KappaPI 2

La base de conocimiento sobre los problemas causantes de ineficiencias se puede ver como un catálogo de problemas habituales de las aplicaciones de paso de mensajes. En el catalogo usado por nuestra herramienta (ver figura 4.3) hemos introducido una serie de problemas disponibles en la literatura de prestaciones

[Fah00, Wol00], así como nuestra base previa introducida en la primera versión de la herramienta KappaPI [Esp98, Esp00], más desarrollos propios en otros problemas. Cada uno de estos problemas, incluye su definición propia en el lenguaje de especificación de KappaPI 2, los métodos de análisis y emisión de sugerencias diseñados para la herramienta. El catalogo de la base de conocimiento de problemas incluye patrones de ineficiencia relacionados con comunicaciones punto a punto, comunicaciones colectivas, sincronización, y detección de ineficiencias de estructuras de programación.

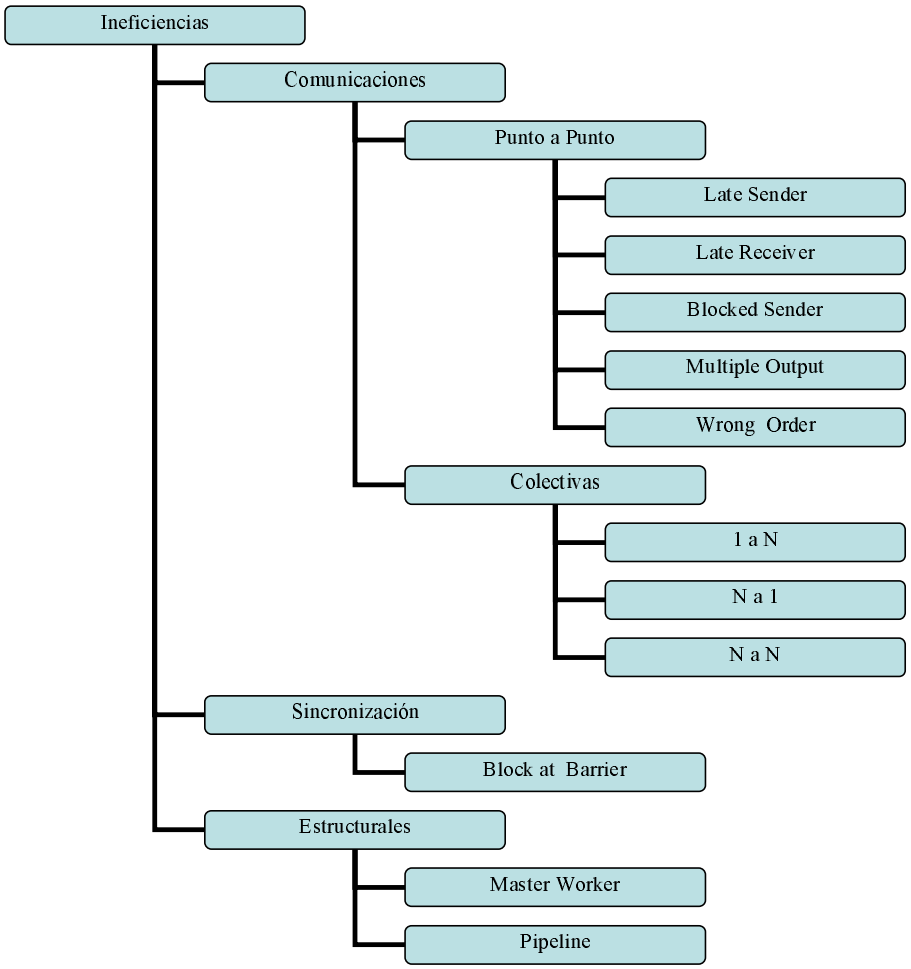

Figura 4.3: Catalogo de ineficiencias en KappaPI 2

En el caso de las comunicaciones punto a punto nos encontramos con los problemas (acompañamos con una breve descripción de cada uno):

Late Sender Aparición tardía de la operación de emisión de un mensaje en una comunicación punto a punto, cuando el receptor ya había iniciado su opera-

ción de recepción.

Late Receiver El receptor se presenta su operación de recepción a posterioridad de la aparición de la operación de envió por parte del emisor.

Blocked Sender Una operación de recepción no puede iniciarse, porque el emisor que la complementa está bloqueado por una operación previa de recepción de mensaje de un tercer participante.

Multiple Output Un emisor emite una secuencia repetitiva de mensajes punto a punto hacia varios receptores.

Wrong Order Emisor y receptor de una comunicación punto a punto, no siguen un orden correcto preestablecido de la emisión de mensajes, siendo el orden diferente al esperado por los participantes en la comunicación.

Respecto a las comunicaciones colectivas, pueden aparecer dependiendo de las primitivas utilizadas (operaciones de tipo scatter, gather, reduce, all), tres tipos de problemáticas, dependiendo de las tareas participantes, y el patrón colectivo (1 a N, N a 1 o N a N) que establece el tipo de comunicaciones:

Early root en operaciones 1-N Operaciones (como scatter) donde depende el avance colectivo del momento de aparición de la tarea con el rol de root. Si el retraso es significativo respecto unas o todas las tareas, la operación global se penaliza con una serie de retardos.

Early taks en operaciones N-1 En operaciones que supongan recolección de datos (como gather, reduce, ...) la aparición tardía de algunas tareas participantes causa retardos a la operación global.

Early task en operaciones N-N Cuando todas las tareas están implicadas con el mismo rol dentro de la operación colectiva, el retraso de algunas de ellas impide el avance global de la operación.

Finalmente con respecto de la sincronización, en las operaciones típicas de espera en operaciones de Barrier, nos encontramos con:

Block at Barrier Donde las tareas que se presenten están esperando a la ultima que aparezca para avanzar.

Por ultimo se detectan ineficiencias basadas en parámetros de estructuras de paradigmas. donde se intenta capturar el modelo del paradigma de la aplicación, o de parte de ella, en una fase de ejecución. Se determinan los parámetros del paradigma, y se tendrá que analizar si existen ineficiencias causadas por un comportamiento incorrecto del paradigma basándose en un modelo de prestaciones de este. En la herramienta se proveen detección y análisis para dos estructuras comunes, Master Worker y Pipeline.

4.7. Detección de los problemas

La detección de problemas de prestaciones de las aplicaciones en sus trazas de eventos por medio de patrones de ineficiencia se ha demostrado como un método valido de generar información automática hacia el al usuario sobre las prestaciones de la aplicación [Jor02, Wol03, Bha05]. Básicamente este conjunto de técnicas, nos permite reconocer estados de espera (tiempo perdido) entre eventos individuales desplazados en el tiempo entre múltiples tareas.

Un concepto que apareció interesante, para ser utilizado en el proceso de detección, era el de clasificar el conocimiento que se dispone en la base de conocimiento, de alguna forma que nos permitiese utilizarlo de forma más eficiente, y nos aportara prestaciones adicionales. En muchas herramientas automáticas (como por ejemplo Expert [Wol03]), el conocimiento especificado se mantiene únicamente como una lista de problemas jerarquizada por tipos, y básicamente se aplica está lista con un problema detrás de otro, para tal de detectarlos, sin ninguna idea de clasificación del conocimiento de los problemas, o sus posibles interrelaciones.

Respecto a esta idea, nos encontramos con conceptos que se usan en determinados algoritmos de aprendizaje por computadora, en los que suele ser habitual clasificar el conocimiento por conceptos [Lan96], y en particular hay una serie de algoritmos basados en lo que se llama árboles de decisión [Lan96, Mit97]. Estos algoritmos están enfocados en los procesos de aprendizaje, y en la adquisición de conocimiento de este proceso, para después usarlo para inferir datos de las observaciones basadas en el conocimiento entrenado a partir de la experiencia. Los árboles de decisión clasifican instancias de conceptos, por medio de ordenarlas en árbol hacia abajo desde la raíz, hasta nodos hoja, lo que automáticamente nos proporciona una clasificación de la instancia. Cada nodo del árbol especifica un test de algún atributo de la instancia, y cada rama descendiendo desde el nodo corresponde a uno de los posibles valores para el atributo. Cada instancia es clasificada por medio de comenzar en la raíz, testeando los atributos especificados en el nodo, moviéndose en el nodo hacia abajo dependiendo del valor del atributo en el ejemplo dado por clasificar. Este proceso se repite para cada subárbol con raíz en el nodo que nos encontremos.

En nuestra herramienta se planteaba el mismo problema, como usábamos la información especificada sobre los problemas, y como la clasificábamos para usarla luego en la detección de los problemas de la aplicación. Y en particular ¿como conocemos que grado de separación lógica tienen los problemas? es decir, son realmente problemas disjuntos, o se pueden ¿combinar o unir de alguna manera concreta?. Estos datos no son fáciles de obtener en un principio directamente de la base de conocimiento de los problema, y se hacia necesario disponer de métodos para clasificar este conocimiento, para mejorar su aplicabilidad.

En nuestro caso tomamos una aproximación (de clasificación inicial del conocimiento, y posterior uso para la detección) que denominamos ”árbol de decisión”, el concepto descrito anteriormente se vuelve diferente, ya que partimos del conocimiento ya desarrollado (por expertos), el cual normalmente nos viene en forma de

jerarquías conceptuales de problemas (relacionadas como vimos con las diferentes primitivas del paso de mensajes para comunicaciones). El problema se relaciona con la clasificación de este conocimiento, en forma patrones estructurales de ineficiencias, en una forma valida para la detección de estos, y que nos permita usar conocimiento en principio no explicito en la especificación.

Cada una de nuestras especificaciones de los problemas, como vimos, nos permite incorporar un problema describiéndolo como un patrón estructural de eventos presentes en el problema concreto, sus restricciones temporales y espaciales, y la evaluación de índices que nos indican su importancia como causantes de ineficiencias.

Los mecanismos, de la herramienta KappaPI 2, para la detección de los problemas están basados en la búsqueda de la estructura descrita en los patrones de los problemas, en la descripción en forma de traza obtenida por la ejecución de la aplicación. Posteriormente una vez identificados, serán clasificados por los índices calculados respecto a las observaciones globales obtenidas de la aplicación, y a las ineficiencias causadas de forma local o global a las tareas de la aplicación, así como su grado de aparición temporal y localización dentro de la aplicación.

KappaPI 2 se inicia con la lectura de la base de conocimiento, y organizándola en el árbol de decisión antes descrito, para posteriormente usar dicho árbol para la búsqueda dentro del dominio de la base de conocimiento, y permitir detectar los problemas aparecidos detectando en la traza de la aplicación las instancias de estructuras de problemas. Dentro de este proceso de detección nos encontramos con las dos fases principales: Construcción del árbol de decisión, y uso como árbol de búsqueda para la detección.

Para la construcción del árbol de decisión, tenemos que tener en cuenta que cada especificación de un problema incluye un conjunto de eventos que define su estructura, que tendrá que ser correlacionada con la traza de la aplicación. Cada problema dispone de un evento característico que debe aparecer denominado evento root, y una serie de eventos relacionados (que han de estar presentes). El árbol de decisión se construye a partir de la lista de problemas, insertando uno a uno estos problemas de manera que cada uno de los problemas puede verse como un camino particular dentro de este árbol. Así un problema determinado es un camino en el árbol desde un nodo raíz hasta un nodo particular hoja.

Durante esta construcción hay una serie de puntos que hay que tener en cuenta:

- Es posible que varios problemas compartan un conjunto inicial de eventos comunes, o sea disponen de caminos parciales iguales en el árbol.

- Un problema puede ser una extensión de otro. En este caso, el problema tiene un nodo final que indica que el problema se ha detectado y por tanto podemos proceder a la evaluación de sus índices. Pero si hay un problema que extiende al actual, el camino puede continuar con la presencia de nuevos nodos (eventos) que amplíen el problema. Por tanto el análisis del problema concreto no puede darse como finalizado, ya que pueden presentarse eventos que extiendan el primer problema hacia el segundo problema. Si finalmente

el segundo problema completo es detectado, podemos calcular sus índices, y el primero puede ser eliminado como problema.

- En varios problemas pueden darse situaciones de un número indeterminado de eventos del mismo tipo. Para solucionar esto es necesario crear nodos en el árbol que representen la ocurrencia de 1 o más eventos del mismo tipo.

En el árbol de decisión, cada problema será visualizado como un camino en el árbol. De manera que posteriormente, a la hora de realizar la correspondencia los eventos encontrados en la traza de la aplicación sobre el árbol nos definirá la detección de un problema. En el camino sobre el árbol, cada nodo tiene una instancia de evento del problema, la posición en el camino define el orden de los eventos en la traza (orden que se explicita mediante las restricciones en la especificación del problema).

Dentro del árbol, teniendo en cuenta los puntos anteriores, nos encontraremos con tres tipos diferentes de nodos:

1. Root como nodo inicial con la instancia del evento root de la especificación.

2. Nodos intermedios como instancias intermedias de eventos relacionados presentes en la especificación.

3. Nodos *End* los cuáles definen la correspondencia completa de un problema detectado, y donde las computaciones, incluidas en la especificación, son evaluadas.

Del tipo de nodo End, nos encontramos con dos versiones diferentes, los nodos *Final* y los nodos *Leaf*. Las diferencias entre ellos, están relacionadas con el camino parcial compartido entre varios problemas. Por ejemplo, examinando un caso simple (ver figura 4.4), encontramos envueltos los problemas Late Sender (LS) y Blocked Sender (BS) con una parte del camino compartido, ya que un BS puede ser inicialmente percibido como LS, sino disponemos del resto de eventos observados.

En este caso (figura 4.4), si hemos llegado al nodo end del LS (en este caso de tipo final), podemos considerar que este problema ha sido detectado, pero tenemos que tener en cuenta no parar en este punto, si más eventos relacionados con el problema de BS llegan a aparecer. En este caso mantendremos el problema parcial (LS) detectado hasta que podamos contrastar si el camino puede ser seguido o no. En el caso que si, y lleguemos al nodo end del problema BS, este nodo (si no hay posibles continuaciones, en nodos descendientes) sera considerado como nodo leaf, y podrá ser incorporado a la lista de problemas detectados que han ocurrido durante la ejecución de la aplicación.

Cuando ya disponemos del árbol de decisión, podemos comenzar la búsqueda de los problemas, por medio de la lectura de los ficheros de traza obtenidos de las tareas de la aplicación en ejecución. Durante este proceso, se creara dinámicamente un modelo de la aplicación y del sistema físico (máquinas, nodos usados) de manera, que sea posible relacionar eventos con tareas y con nodos en los que se

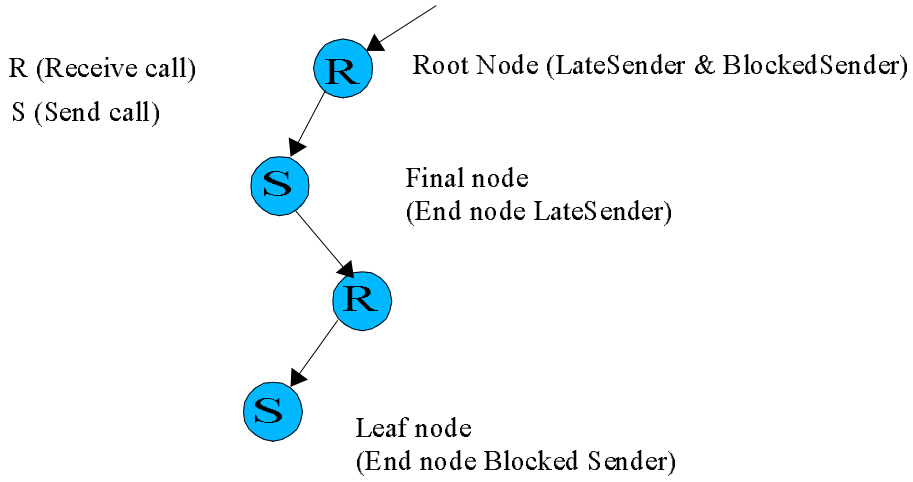

Figura 4.4: Caminos compartidos entre problemas de prestaciones

encuentran ejecutando, además de poder tomar métricas y estadísticas diversas sobre las llamadas utilizadas, tiempos, y localización física de las llamadas. En este punto, es importante tener presente que las trazas (y por tanto las herramientas de traza usadas) han de ser capaces de guardar información, de la localización de las llamadas de paso de mensajes producidas, incluyendo información como la linea de código fuente, y el fichero de código fuente donde se encuentra.

El motor de detección de problemas comienza así leyendo los eventos de traza, que se encuentran ordenados por su estampa (*stamp*) de tiempo global. Cada evento se intenta corresponder dentro del árbol de decisión, intentando moverse bien desde la raíz del árbol, si no hay caminos de problemas abiertos en ese momento, o bien intentando continuar el evento como evento siguiente en el árbol de los problemas que se encontrasen abiertos, o aquellos finalizados pero con posibles extensiones por abrir. Si el nodo se detecta, como iniciador (o continuador) de un problema determinado, se marca como nodo incluido en el problema para continuar a partir de el. En el caso que un nodo hoja final se alcanza, los índices son evaluados y el problema es incluido en la tabla de problemas detectados para su posterior clasificación. El problema detectado, y guardado en la tabla, incluye toda la información (eventos, tiempos, tareas, posición en código relacionado) que es necesaria para relacionarlo con la ejecución de la aplicación, así como en particular con los lugares en el código donde se produjeron las llamadas de paso de mensajes relacionadas.

Durante el proceso de correspondencia de los eventos de traza con los problemas formando la base de conocimiento incorporada en el árbol de decisión, existe otro problema relevante que tratar, debido a los *aliasing* de eventos. Este problema es debido a los problemas de coherencia de nombres entre la especificación y la correspondencia con los eventos encontrados en la traza. En cada problema

descrito, el usuario utilizara para la especificación una serie de nombre simbólicos para los eventos, utilizándolos para especificar las instancias, las restricciones y los cálculos a realizar. En el proceso de correspondencia entre los eventos de traza y el árbol, se realizan intentos de realizar la correspondencia de cada evento con un nombre simbólico. Si se establece esta, el evento será incluido en la correspondencia parcial del problema, y en una tabla de símbolos con el par (nombre simbólico evento, instancia evento). Este proceso de asignación de nombres (aliasing), es complejo debido a los diferentes eventos que podemos encontrarnos dentro del conjunto de eventos presentes en un problema. En un problema concreto se presentan un subconjunto de eventos, en los cuáles está establecido un orden casual, pero otros se presentan sin orden (es necesaria la presencia de estos eventos, pero no existen restricciones temporales). Este subconjunto de eventos (sin restricciones temporales) obliga a generar combinaciones de diferentes posibles asignaciones de nombres, que habría que validar, y hacer/deshacer, según las nuevas presencias de eventos que aparezcan.

La aproximación del árbol de decisión, permite una flexibilidad importante, para la expresión de la búsqueda y detección de los problemas especificados como ramas finales del árbol, está aproximación es novedosa [Jor02, Mar04], pero es interesante destacar otras aproximaciones similares (con orientación final diferente): a) Técnicas de entrenamiento / aprendizaje de herramientas de prestaciones bajo benchmarks [Vet00] para crear un árbol de decisión, o bien b) la introducción de técnicas semejantes en otros campos como el análisis de prestaciones de bases de datos [Dia05].

4.8. Clasificación

En muchos casos, la clasificación de los problemas detectados no es trivial, ya que estos dependen fuertemente del modelo de prestaciones usado, y de los índices usados en su detección (muchas veces contrapuestos entre si).

En [Fah00] se sugiere la noción de propiedad de prestaciones (Performance property), que caracteriza un comportamiento especifico de prestaciones, que se convierte en un problema de prestaciones (performance problem) cuando se superan ciertos limites establecidos respecto al impacto de otros problemas, o bien a limites impuestos por el usuario. Pero cabe destacar que en muchos casos el usuario (no experto) es imposible que tenga una idea clara, o pueda dar limites aceptables o adecuados.

En nuestro caso, no utilizamos limites proporcionados por el usuario, sino que pueden ser bien proporcionados en forma de conocimiento incluido en los problemas, o ser calculados de forma automática para problemas semejantes según las medidas tomadas, y los índices que se utilicen para la clasificación de los problemas.

En estos índices pretendemos que la tabla de problemas detectados sea filtrada, mediante clasificación por índices que evalúen el impacto de los problemas. Parte

de este proceso ya se realiza automáticamente durante el registro de los problemas detectados, debido a que hay una representación única para un problema, dada su localización entre tareas, y código fuente. De esta manera un problema descrito dentro de un conjunto de tareas, con regiones de código (lineas y ficheros fuentes) iguales, es almacenado de forma acumulada, de manera que registramos su aparición y su repetición, así como las diferentes medidas de índices (calculados), y tiempos de ocurrencia de cada instancia del problema.

El objetivo final es filtrar un subconjunto de los problemas presentes que genere el mayor impacto de ineficiencia. El impacto es evaluado en términos de localidad (causa ineficiencias en un conjunto reducido de tareas, o en la aplicación global), y impacto temporal (grado del impacto). Otros aspectos a evaluar son la repetición, o el solapamiento temporal con otros problemas diferentes y el impacto resultante.

Cada problema encontrado durante la fase de detección es añadido en la tabla, teniendo en cuenta que las repeticiones son acumuladas bajo la misma definición de problema, en base a las tareas participantes en el problema, y la localización en código común. En cada problema se almacena:

- Identificación (teniendo en cuenta localización en tareas y código).
- Lista de tareas que participan en el patrón del problema.
- Número de repeticiones (instancias) del problema encontradas.
- Lista de instancias del problema: Para cada instancia del problema, se almacenan los eventos correspondientes que han sido mapeados desde la especificación del patrón de ineficiencia a los eventos en la traza. Incluyendo toda la información de los eventos.
- Tiempos de cada instancia: Cada una de ellas incluye un tiempo de inicio del problema, correspondiente al primer evento participante, y un tiempo de finalización, correspondiente a la salida del último evento. Esto nos permitirá para cada problema estudiar los posibles solapamientos con otros problemas.
- Tiempos idle: En cada instancia se incluirá la lista de los tiempo idle que se hayan producido, en las diferentes tareas que forman parte de la especificación del patrón del problema. Permitiendo acumular estos tiempos por tarea para identificar las tareas más perjudicadas, por la repetición de la aparición del problema. Así como permitir examinar la distribución de tiempos idle entre las tareas participantes, por ejemplo en el desarrollo de operaciones de comunicación colectivas.

Con todos estos datos podemos afrontar esta etapa de clasificación mediante el uso de diferentes técnicas:

- Clasificación por acumulación de aparición, en el caso del mismo problema repetido.

- Clasificación mediante índices, que nos permita evaluar la importancia del problema, según los datos que hemos recolectado de tiempos.
- Tratamiento de uniones y/o posibles solapamientos de los problemas, para determinar si dos o más problemas están interrelacionados, y utilizar medidas para decidir cual de ellos es el de mayor importancia, y/o causante del resto.

En la siguiente figura (fig. 4.5) observamos, el desarrollo de una serie de tareas, en donde aparecen una serie de repeticiones (instancias) del mismo problema (en este caso un late sender) a lo largo de la ejecución, con sus tiempos perdidos asociados:

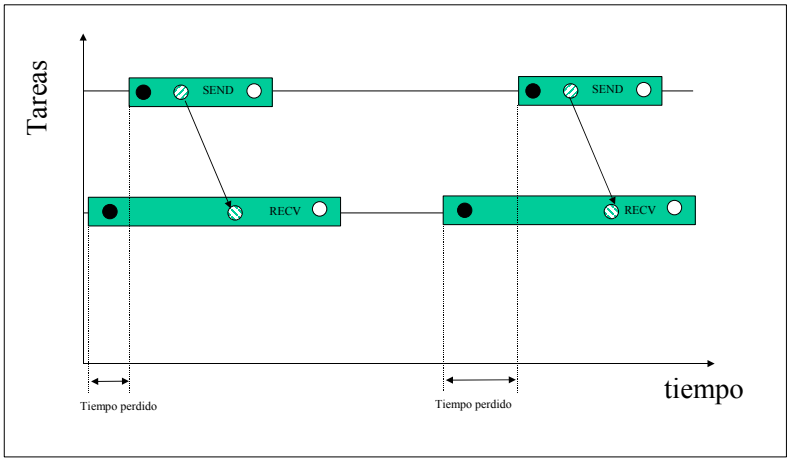

Figura 4.5: Repetición durante la ejecución de un problema

Estos tiempos de intervalo, nos permiten ver si se producen (para diferentes problemas) solapamientos entre ellos. Pudiéndose dar entre dos (o más) problemas, tres situaciones:

- No solapamiento, lo cual nos indica que los problemas no tendrían una causa efecto directa.
- Solapamiento parcial, los problemas se solapan parcialmente en el tiempo. Pudiendo indicar dos situaciones diferentes: a) los dos problemas no comparten el conjunto de tareas, indicando que el problema forma parte de dominios diferentes de tareas dentro de la aplicación. b) O comparten parcialmente (o

totalmente) las tareas participantes, en donde se puede producir una relación causa efecto entre los problemas.

- Inclusión: Los limites temporales de un problema se incluyen en los limites del otro. Debido a la construcción propia del árbol de decisión usado para la detección, esta situación en problemas especificados se incluiría, ya que el problema formaría parte de un problema mayor que seria detectado. Esto siempre que realmente, no fueran dominios disjuntos de tareas, los que se involucran en cada tarea.

En la figura 4.6 observamos los diferentes casos que podemos encontrar, teniendo en cuenta dos posibles problemas (P1, P2) cuyas apariciones afectan a una zona espacio-temporal de la ejecución, teniendo en cuenta las tareas presentes (lineas horizontales representando el conjunto de eventos de su traza), y los recuadros P1 y P2, conteniendo la zona afectada (tareas, y pedazos de traza afectados). Cada problema P1 y P2, dispone de un rango de tiempo activo, desde el inicio del evento inicial del problema al ultimo que pertenece. Los tiempos se comparan en los dos problemas, para ver cuales casos anteriores estan interveniendo. En los casos de la figura, no hay solapamiento de tareas afectadas

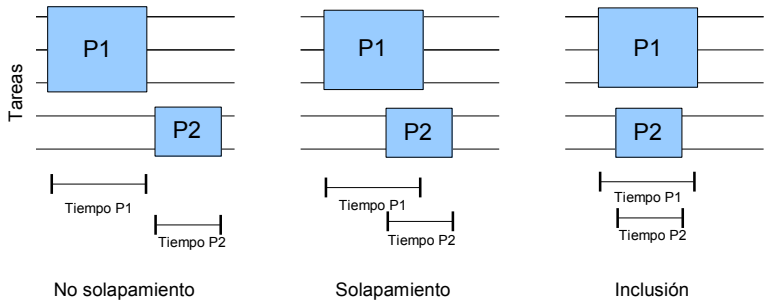

Figura 4.6: Escenarios en la aparición de problemas

En general durante la ejecución de N tareas, durante un tiempo T de aplicación, se presentaran diferentes situaciones de las mencionadas, que tendremos que tener en cuenta a la hora de la clasificación de tiempos, e importancia relativa de los problemas encontrados.

Después de la clasificación acabamos con una fase de análisis, donde intentamos encontrar que causas han provocado la aparición de estos problemas con gran impacto en las prestaciones de la aplicación.

4.9. Análisis de causas: Casos de uso

Una vez el problema concreto ha sido detectado y clasificado, es necesario llevar a cabo el análisis que nos permita determinar su causa principal de aparición, y poder determinar medidas para eliminarlo (en el mejor de los casos), o bien reducir sus efectos en las prestaciones al mínimo, mediante la emisión de ciertas sugerencias hacia el desarrollador para que actúe sobre las causas detectadas.

Este tipo de análisis se realiza mediante el uso de una segunda incorporación de conocimiento en la herramienta, a propósito de los casos de uso que puede presentar el problema.

Básicamente cada problema de prestaciones, puede haber ocurrido por una serie de casos diferenciados, que normalmente necesitaran un examen detallado de los tiempos involucrados, así como de su localización en el código para poder llegar a una conclusión. En este punto, hay que recordar, que necesitamos que los eventos de la traza incluyan la información concerniente al código fuente, en forma de lineas y ficheros fuente donde se localiza la ocurrencia del evento. En este caso, tal como previamente hemos mencionada, nos hemos servido de herramientas de traza como TapePVM para el entorno PVM, y de nuestro desarrollo de MPITracer para el entorno MPI (MPICH) para disponer de eventos con esta información.

Este análisis de causas usara un nivel de especificación de conocimiento en forma de plantillas (*templates*), que nos permitirá especificar los casos, las condiciones a examinar en base a la información de las fases anteriores, y acciones para obtener nueva información suplementaria según el caso, normalmente mediante análisis de código fuente en etapa posterior. El análisis de causas ha realizar dependerá del problema detectado, tanto este análisis como la posterior construcción de sugerencias, se hará a partir de especificaciones, disponibles en ficheros, relacionadas con la especificación inicial del problema que la herramienta kappaPI 2 ha cargado en el inicio de su ejecución

El análisis del problema, así, puede especificarse como un fichero de plantilla con un formato determinado (utilizaremos también un formato XML), en el que se especifican casos del problema a testear en un formato if-then-else, junto con las condiciones que los validan, y posibles actuaciones extras a realizar, o análisis adicionales para validar el problema.

En cada caso las condiciones deberán poderse validar bien con la información ya disponible en la tabla de problemas detectados (su información relacionada) o con un análisis de código complementaria. Cada caso puede incluir algunas relaciones con el código que habrán de ser verificadas. Estas relaciones, y tests en el código fuente, serán especificadas como llamadas a una pequeña API disponible en la herramienta para análisis de código fuente.

Finalmente cada caso ha de ofrecer una recomendación final que sugerir de cara al usuario sobre la forma de impedir, reducir el impacto del problema en la aplicación. En este caso, dado el enfoque final hacia el usuario, esta recomendación (*hint*) se dará en forma de lenguaje natural, por tanto se usaran mensajes predefinidos que permitirán incluir los parámetros, e informaciones, del problema

necesarios para situar al usuario con relación a que zonas del su aplicación se ven afectadas por su problema. En el caso de utilización de una interfaz de usuario podemos usar los mecanismos de vistas necesarios para indicar claramente al usuario que lineas y ficheros se ven afectados.

4.10. Especificación del análisis

En las siguientes subsecciones, observamos cada uno de los problemas de nuestro catalogo, para determinar los casos de uso posibles que se nos pueden presentan, y de esta forma poder deducir las actuaciones que se podrían producir, en función de la información disponible, y por tanto las sugerencias que podremos dar de cara al usuario final.

4.10.1. Casos de uso

Como hemos comentado cada problema presentara un listado de casos de uso, en forma de condiciones que se han podido producir de manera que causasen la instancia concreta del problema. Según el análisis de estos casos podremos deducir la estructura de la instancia del problema, podremos pedir información adicional para refinar el análisis, y podremos emitir la sugerencia final.

4.10.2. Punto a Punto

En el caso de las comunicaciones punto a punto, surgen varias tipologías de problemas:

- Relacionados con las dos tareas participantes, de manera que en particular, con la existencia de las comunicaciones bloqueantes, se producen tiempos significativos de espera para que se produzcan las comunicaciones. O bien el retraso o avance de alguna de las tareas en su realización, nos depara retardos extra.

- Con posible relación con más tareas, cuando una operación de comunicación punto a punto, se ve retardada, o perturbada por terceros. De manera que bien su iniciación se retarda, o su finalización.

- Las comunicaciones punto a punto, pueden estar englobadas en una estrategia mayor de comunicación, entre las tareas participantes, pero el diseño de la estrategia, o la puesta en práctica no es adecuada para obtener las mejores prestaciones.

En nuestro caso particular, los problemas relacionados con comunicaciones punto a punto, son observados generalmente desde operaciones de recepción bloqueante, que son las que causan los mayores tiempos de espera. Aunque hay que

tener en cuenta que en caso de las no bloqueantes podemos encontrarnos con esquemas semejantes, dependiendo de si utilizamos estrategias (para completar la operación de recepción) basadas en test iterativos, o bien por espera activa.

4.10.2.1. Late Sender, Late Receiver

En el caso del Late Sender, la operación de envío por parte de la tarea emisora se ha producido, con posterioridad a la aparición de la operación de recepción en la tarea receptora. La consecuencia es el bloqueo de la tarea receptora, hasta que pueda iniciarse la operación cuando aparezca la la operación send, y pasado su tiempo de startup, pueda iniciarse la transmisión real. En este problema el receptor habrá permanecido en estado idle entre el inició de las dos llamadas, perdiendo tiempo que podría haberse usado en computación útil.

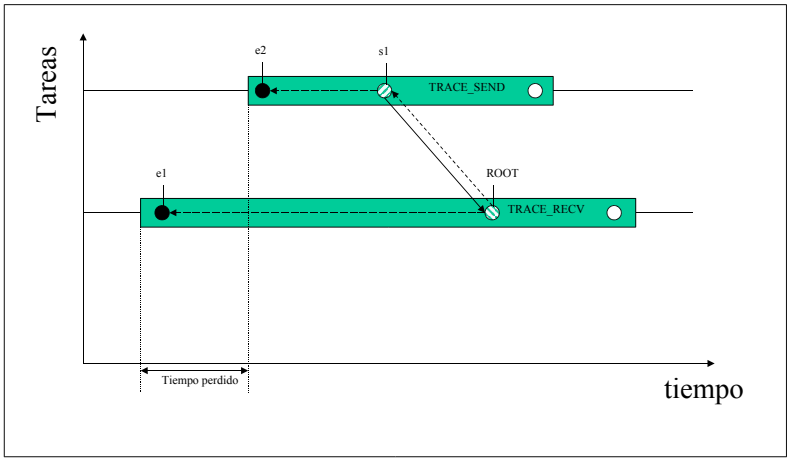

Figura 4.7: Ineficiencia causada por Late Sender

Así en este caso podemos observar que el tiempo perdido, aproximado (o digamos medible en traza), sera desde el tiempo del evento de inicio de la operación de recepción, hasta el tiempo de inició de la operación de envío.

En el Late Sender podemos llevar el análisis de casos de uso, a partir de las observaciones respecto de los eventos participantes (recv previo, y send posterior):

- Examinar posición del evento de send, la primitiva de envío utilizada, para comprobar la existencia de dependencias de datos en sus parámetros, si tales dependencias no existen o pueden espaciarse, podemos intentar mover el

código de la primitiva de send, para que se realice antes, subiendo su código tanto como se nos permita, una serie de lineas antes. Se le avanza el tiempo su ejecución, y disminuyendo así la espera en el receptor.

- En el caso que existan dependencias, hará falta reorganizar el código: intentar avanzar cálculos, romper envíos de mensajes en algunos más simples que puedan enviarse antes (avanzando las comunicaciones, permitiendo que el receptor reciba antes).

- Otra posibilidad, es utilizar semejantes análisis para el código del receptor, podemos examinar si la primitiva de recepción se puede retrasar en el tiempo (avanzar en las lineas de código), por ejemplo mediante la colocación de cómputos independientes de la recepción, con anterioridad a la primitiva.

- Si las dependencias impiden cualquier movimiento de código, podría sugerirse un cambio de mapping de las tareas, si nos encontramos en un entorno heterogéneo, que se adaptase mejor, a la velocidad de emisor y receptor.

Veamos con un ejemplo de Late Sender como la herramienta, realizaría este análisis de casos de uso. Durante la fase de detección, se habría encontrado un problema de Late Sender, que se habría clasificado teniendo en cuenta su posición en código, así como los índices de importancia relativa del problema, y el número de sus apariciones, así como el tiempo acumulado de retardo que habría causado a la tarea receptora.

En la fase de análisis de los problemas, en la arquitectura abierta propuesta para la herramienta KappaPI 2 tenemos una segunda fase de incorporación de conocimiento, en forma de especificación abierta de los casos de uso y las posibles sugerencias que se pueden realizar a traves de la información obtenida y analizada. Proporcionaremos para el conjunto de nuestros problemas, una especificación donde en cada problema procedemos a listar una serie de casos de uso en forma de tests que tenemos que verificar para que se cumpla el caso del problema, y acciones que podemos proponer obtenidas del análisis de código que se realizará (que comentaremos en las próximas secciones).

Por ejemplo, en el caso del Late Sender, nos podemos encontrar con la siguiente especificación de análisis (fragmento):

Especificación del anàlisis

```
1  <ANALYSIS name="Late Sender">
2    <points>
3          <place name="recv" event_info="E1">
4          <place name="send" event_info="E2">
5    </points>
6    <test type="CODE_UP" place_name="send">
7      <message>Hint: Receive delays</message>
8      <hint_true>
9        Code in send call can be located before
10       this place, no data dependences
11     </hint_true>
```

```
12      <hint_false>
13        Try to refactor this code, can be located
14        before this place
15      </hint_false>
16    </test>
17    ...
18  </ANALYSIS>
```

En este especificación:

- Relacionamos el análisis con el nombre del problema (*Analysis name*), que será el que se uso durante la especificación estructural del problema, utilizada durante la fase de análisis.

- Los puntos (*points*) son los lugares que se usaran durante el test, y posterior análisis de código, damos nombres a estos puntos, y los relacionamos con la información (en forma de eventos) que están almacenados de la fase de análisis.

- El *test* representa el caso de uso, planteamos una situación que observamos (*message*), realizaremos un test en el código (*type*), en el caso de que pueda realizarse, emitiremos una recomendación genérica (*hint_true*) más la información detallada que se obtuviera del análisis de código propuesto. En caso de no validarse el caso, podemos continuar con las validaciones de los otros casos de uso, o bien proponer una alternativa genérica (*hint_false*), con información detallada de la localización (por ejemplo región de código a examinar o refactorizar).

En el caso del problema Late Receiver es semejante al anterior, pero invirtiendo los términos de aparición de las operaciones. Aunque este caso, es más raro debido a la propia implementación de la operación send que se realiza en los diferentes entornos de paso de mensajes. Sólo si la implementación, o bien una llamada explicita de send, es en modo síncrono, puede producirse este problema. En algunos casos también puede influir el tamaño del mensaje, si este supera ciertos limites la comunicación puede convertirse en síncrona, para no agotar los recursos de memoria en reserva de buffers para la comunicación.

El tiempo perdido, en el caso de observarse el problema, se producirá sobre la tarea emisora, desde que inicie su operación, hasta que se presente la operación de recepción.

En los casos del Late Receiver deberemos observar:

- Examinar posición de la llamada en el receptor, si hay dependencias respecto lo que se recibirá, o podemos colocar su código más atras (en números de linea menores), adelantando en el tiempo su llamada, ya sea en el código del bloque, o en la función/procedimiento donde se encuentra.

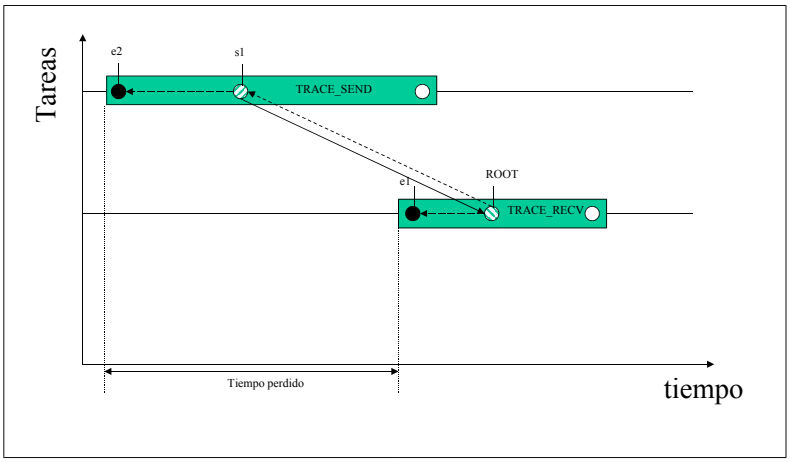

Figura 4.8: Ineficiencia causada por Late Receiver

- Igualmente podemos actuar en el otro lado de la comunicación retrasando el emisor, si este puede realizar algunas computaciones no dependientes en datos.

- Si esto no es posible, sera necesaria una reorganización de código más profunda, o alterar el patrón de comunicación, por medio de reorganizaciones de código, avanzar o retrasar cálculos, o partir la comunicación en varias más simples que permitan un inicio más rápido en el receptor, por ejemplo recibiendo únicamente aquellos datos imprescindibles, que le permitan continuar. Y dejando para envíos posteriores los demás.

4.10.2.2. Blocked Sender

En el Blocked Sender, hay una participación de un mínimo de tres tareas, en las cuáles una comunicación punto a punto, entre dos tareas ve su emisor bloqueado, debido a que está previamente esperando por un bloqueo de recepción de una operación previa.

Entre sus causas podemos encontrar: a) una coincidencia de transmisiones de forma puntual. b) puede representar una dependencia de datos de un grupo de tareas, la cual afectara al rendimiento.

El análisis de casos dependerá de las dependencias que se observen entre tareas, y las relaciones de los mensajes participantes en las dos comunicaciones existentes.

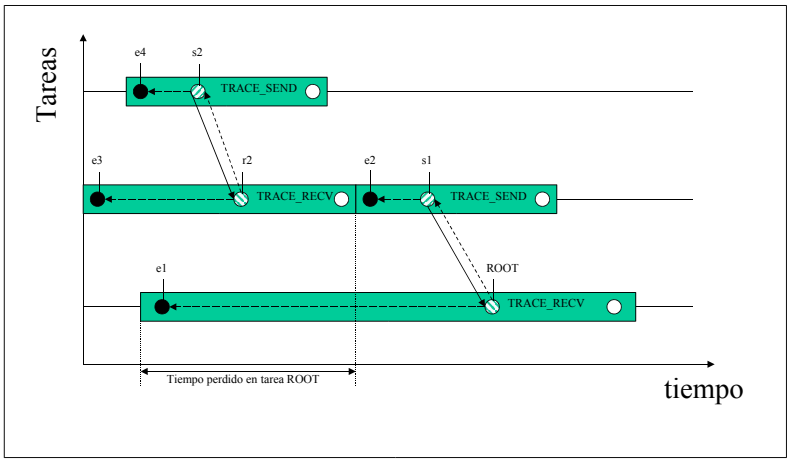

Figura 4.9: Ineficiencia causada por una situación de Blocked Sender

En las sugerencias que podremos emitir hay que tener en cuenta, la recogida de información sobre el emisor, el receptor y los mensajes enviados, y la revisión del código fuente para tener en cuenta las posibles dependencias.

Para este análisis de Block Sender, la problemática principal aparece entre la relación de dependencias, en la tarea que primero recibe y después emite, pudiéndose dar los casos siguientes:

- No hay dependencias entre los mensajes, el mensaje recibido no provoca ningún cambio en los datos del mensaje que se enviara, por tanto podemos reorganizar los mensajes de forma que el segundo sea enviado antes, pudiendo así eliminar las esperas por bloqueo del emisor.

- El mensaje ha enviar se obtiene a traves de cálculos o transformaciones del primer mensaje. Puede entonces intentarse trasladar este cómputo a la primera tarea de manera, que pueda primero enviar el segundo mensaje, y a continuación el anterior primero.

- Son el mismo mensaje, siendo reenviado por la segunda tarea. En este caso podemos realizar operaciones de broadcast (o multicast si está disponible) desde la primera tarea.

4.10.2.3. Wrong Order

Este problema, lo podemos englobar en la categoría de estrategia de comunicación. En este caso dos tareas tienen que realizar una comunicación compuesta por diferentes mensajes, cada uno de ellos es enviado consecutivamente al receptor. En este caso la ineficiencia surge cuando no hay un orden común preestablecido, pudiéndose generar mensajes que no estarán llegando en el orden esperado de consumo por parte del receptor.

Si nos situamos en el peor caso, todos los mensajes pueden llegar en el orden contrario al que se esperaba consumir, causando así una carrera de mensajes almacenados en buffers (y por tanto gasto de recursos, especialmente en memoria), que no liberamos hasta que aparece el último mensaje, que resultaba ser el primero que se necesitaba consumir.

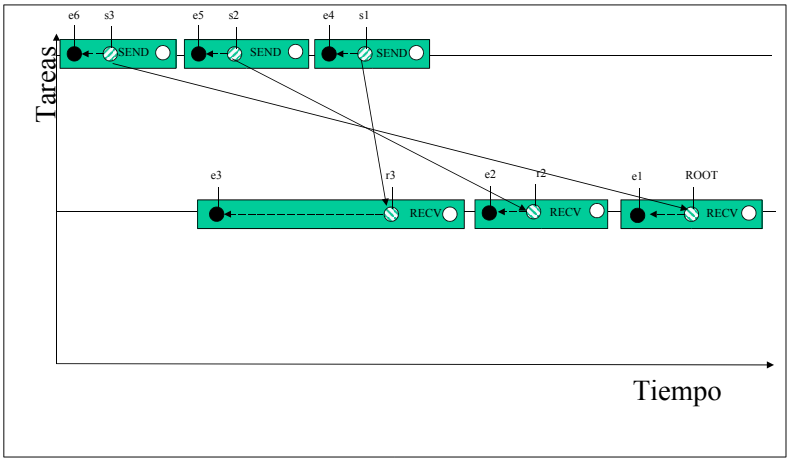

Figura 4.10: Ineficiencia causada por una situación de Wrong Order

Este tipo de problema solo puede darse con casos (por otro lado habituales) de operaciones de envío (send) no bloqueantes. Si se llegara a utilizar sends bloqueantes, el problema seria mucho mayor, ya que estaríamos delante de situaciones de deadlock de las dos tareas, ya que el emisor esperaría al consumo de un mensaje que no se producirá, mientras el receptor espera un mensaje que no se enviara.

En este caso la tarea receptora, sufre un tiempo considerable de espera en función desde su inicio de recepción del primer mensaje, hasta la aparición de este en ultimo lugar por parte del emisor. Además del gasto consecuente en el almacenamiento de los mensajes en transito que aún no han sido recibidos.

Como casos de estudio observar:

- Posibles agrupamientos de mensajes, si los mensajes están en una serie seguida de envíos entre pares emisor-receptor, puede evaluarse la posibilidad de agruparlos, si estos no sufren dependencias.
- Proponer el reorden correcto.

Un caso especial de este problema podría producirse en el caso de utilización de recepción no bloqueante, debido a que tendríamos muchas operaciones de receive iniciadas, que necesitarían bien de estrategias de espera activa (lo que las convertiría prácticamente en el problema inicial), o bien con estrategias de test de operación completa, en un exceso de test para verificar las operaciones que no se podrían iniciar. Aunque en este caso las soluciones también irían en los mismos términos que las propuestas, con el añadido del posible solapamiento de operaciones de test con cómputo, pero de difícil establecimiento por el posible número alto de operaciones no bloqueantes en marcha, y el gran número de combinaciones comunicación-test que se podrían sugerir.

4.10.2.4. Multiple Output

En el Múltiple Output, nos encontramos con un conjunto de tareas que son receptoras de una secuencia de mensajes que se inician en serie desde un emisor común. El envío en serie provoca que las tareas se bloqueen hasta que les llegue el mensaje que les corresponda. Podríamos describirlo como que observamos tareas bloqueadas por mensajes, donde el emisor no ha comenzado la trasmisión aún debido a que está realizando envíos a otras tareas.

Siendo la tarea que reciba en último lugar la que más tiempo habrá esperado (dependiendo en ocasiones de la posición de la aparición de su recepción).

Posibles soluciones: a) si se trata de los mismos mensajes, podemos sugerir el uso de primitivas de broadcast o multicast, dependiendo de las tareas participantes [Mar99]. b) En caso de tratarse de alguna repartición de datos, podemos sugerir primitivas de tipo scatter si los datos pertenecen a algún tipo de estructura regular (o formas con tamaños variables si la utilización no es regular). b) En el caso que se trate de obtener algunos resultados de alguna operación común pueden sugerirse operaciones de reduce, siempre que esta operación sea identificable. c) En caso de que no haya una estructura clara de datos en la repartición, puede sugerirse crearla, y mejorar el patrón de envíos.

4.10.3. Colectivas

En las primitivas colectivas frecuentemente surgen problemas de prestaciones, debido a su propia semántica de participación, que supone inherentes efectos de sincronización entre las tareas participantes [Pje05]. Con la consecuente creación de retardos en algunas (o todas las) tareas, debido a la desviación temporal de aparición de algunas de ellas para participar en la operación conjunta.

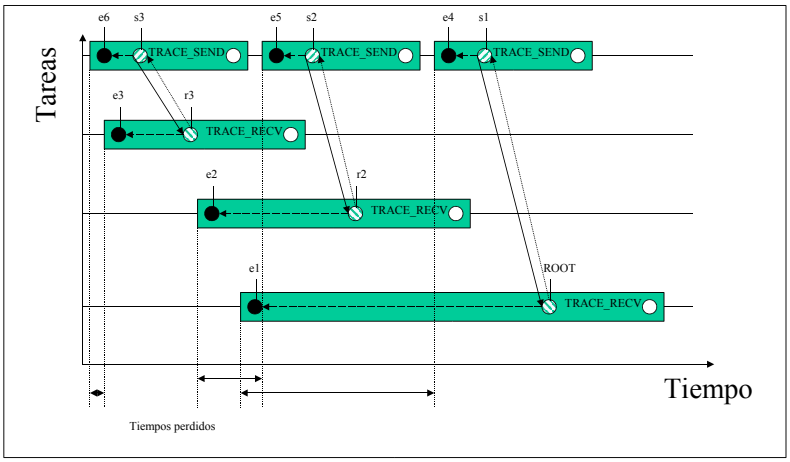

Figura 4.11: Ineficiencia causada por situación de Multiple Output

En algunos casos dependiendo de la dirección de las comunicaciones, el problema se encuentra en la tarea que ejerce el rol de root de la comunicación, decidiendo la eficiencia de la operación su momento de aparición con respecto al resto de tareas. Mientras en otros casos, la distribución de los tiempos de llegada es lo significativo, o simplemente la aparición de la operación en la ultima tarea componente del grupo.

Consideramos las ineficiencias en sincronización, un caso particular de colectiva, que si bien no engloba una comunicación directa de datos (normalmente si a nivel de implementación), si que se sugieren los mismos problemas en cuando a distribución de tiempos de aparición entre las operaciones de las tareas participantes.

En particular también hay que tener presente como aspecto relevante (ya comentado, relacionado con la clasificación de estos problemas. Ya que el número total de las tareas participantes, ya sean en grupos discretos de ellas, o de forma global a todas las tareas de la aplicación, comportan diferentes problemáticas, acentuandose en el caso de colectivas globales, debido al paro global de toda la aplicación.

4.10.3.1. Blocked at Barrier

En este caso de sincronización [Tse02], todo el grupo de (o todas) las tareas han de iniciar la operación, sin distinción entre ellas, de manera que hasta que no lleguen, no pueden continuar. Básicamente, un grupo de tareas aparece bloqueado,

si los tiempos de llegada entre ellas difieren demasiado unos de otros.

Esto nos provoca diferentes tiempos de llegada de las tareas, debido posiblemente ya sea a desbalanceos de carga de trabajo, entornos heterogéneos o a problemas anteriores que afectaban solo a parte de las tareas.

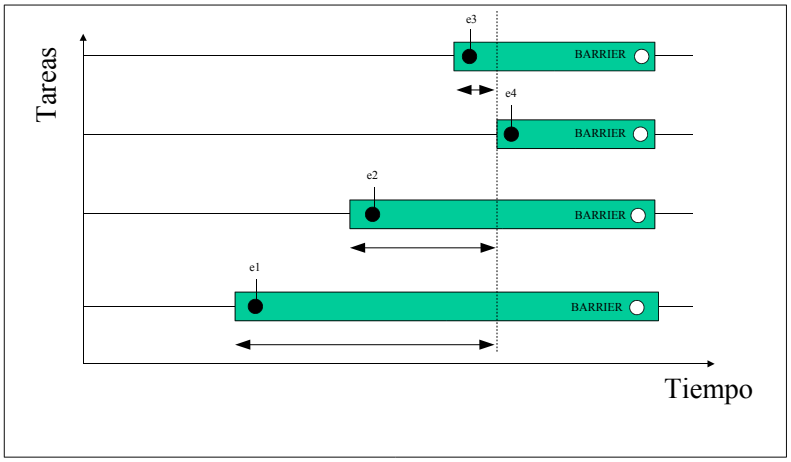

Figura 4.12: Ineficiencia por llegadas a Barrier

El tiempo de llegada de la ultima tarea es la que define con respecto a las otras los tiempos de espera que se producen. En este caso, la primera tarea en llegar alcanzando el Barrier permanece bloqueada hasta que la última haya llegado.

Entre el análisis de casos, tendremos que observar básicamente las diferencias de tiempos en la llegada de las tareas, y observar las conclusiones que se puedan extraer:

- Examinar si estamos las llamadas se producen desde el mismo código. En este caso el desbalanceo, se puede producir por desbalanceos causados por mapping adverso en sistemas heterogéneos. O bien es sistemas homogeneos, ser causado por la existencia previa de problemas, que han afectado a las tareas que han llegado tarde.

- Si las llamadas proceden de códigos diferentes, habrá que examinar si es posible, conseguir desplazar el código de la llamada colectiva hacia atras, si no existiesen dependencias. O como alternativa adelantarlo en las rápidas introduciendo cómputo que no tenga dependencias.

En todo caso, en las primitivas de este caso de sincronización, como en las colectivas posteriores tendremos que examinar los tiempos de diferencias entre la primera y ultima tarea en realizar la operación, así como las desviaciones de tiempo observadas en el resto de tareas. Cuestión que hemos tenido ya en cuenta al acumular la información en la fase de clasificación de los problemas, con diferentes índices de tiempo perdido en las tareas, y el número de tareas que se veían afectadas por ello.

4.10.3.2. Colectivas 1 a N

En este tipo de comunicación se produce la situación, desde una tarea con el rol de root, de envió de mensajes al resto de las tareas. La aparición tardía de esta tarea causa una situación de bloqueo a todas aquellas tareas que hayan llegado antes.

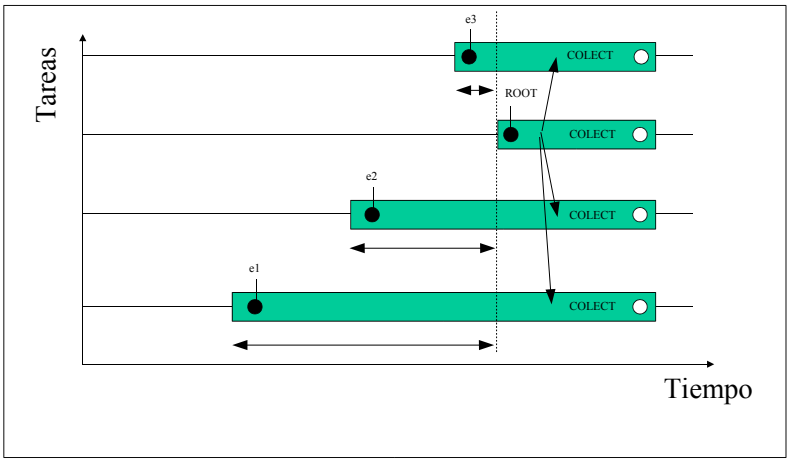

Figura 4.13: Ineficiencia causada por colectivas 1 a N

En el análisis de este problema, nos encontramos con problemáticas referentes a la distribución de los tiempos de llegada de las tareas participantes. Siendo en este caso especialmente problemática la aparición tardía de la tarea emisora, que causara retardos en el resto de tareas que hayan llegado antes. Pudiéndose dar el caso que sea un retardo general de la operación si la tarea emisora llega en ultimo lugar, creando de hecho un efecto semejante al bloqueo delante de una operación de Barrier, creando una sincronización no implícita.

En general podríamos observar los siguientes casos según las distribuciones de los tiempos de llegada:

- Llegada tardía del emisor, causando retardos generalizados a las tareas receptoras. Intentar adelantar en código el emisor, si no existen dependencias. O alterar el mapping de este respecto al resto, si los tiempos de llegada del resto son uniformes.

- Dependiendo de los tiempos de llegada de las tareas receptoras, retrasar la operación en código, si fuera posible adelantar cómputo posterior.

4.10.3.3. Colectivas N a 1

En este problema, el receptor puede sufrir tiempos de espera, si su llegada se produce antes que las de las tareas que enviaran la información, por tanto cuando todavía no hay información disponible.

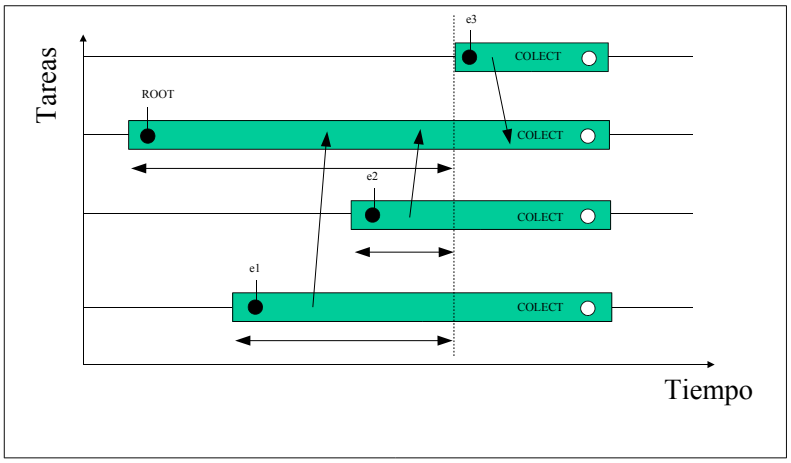

Figura 4.14: Ineficiencia causada por colectivas N a 1

En este análisis los casos son producidos por la relación de la aparición de la tarea receptora (root), respecto al resto de participantes. Pudiéndose dar casos de:

- Aparición temprana del receptor, ocasionando en este grandes retrasos, según la dispersión de las tareas emisoras de datos. Este caso es parcialmente evaluable, dependiendo de la distribución de la llegada de las emisoras: a) Si la tarea receptora llega primero que ninguna, claramente pierde tiempo hasta la aparición de la primera emisora; En caso contrario, una llegada temprana cuando ya existen algunas tareas emisoras, puede considerarse ya tiempo

útil por parte de la receptora. Aunque en este ultimo caso, la ineficiencia la causaran la llegada de tareas emisoras posteriores con diferencias apreciables respecto la llegada de las primeras. También tendríamos que observar dependiendo de la implementación de la librería, si se permiten que algunas tareas emisoras acaben su participación, antes de que lleguen otras.

- En caso de llegada tardía del emisor, estará afectando a todas las tareas, o a parte de ellas, que se presentaron antes causando los respectivos retardos, siempre que por implementación (por ejemplo buffers o mecanismos asíncronos) no se les permita acabar antes de la aparición del receptor.

Respecto a que podemos sugerir realizar, dependerá en cierto grado de la uniformidad del las tareas (modelos SPMD), del mapping heterogéneo o homogéneo, encontradonos con:

- En caso de que se presente el receptor antes, tendremos que comprobar en que grado podemos avanzar en el tiempo los envíos del resto de las tareas adelantándolos en el código, o eliminando (si fuese el caso) problemas previos que lo causaran (debido a solapamiento de los problemas anteriores causando retardos en las tareas emisoras). O por otra parte retrasar en tiempo el emisor, adelantando ciertos cálculos que pudiesen realizarse antes.

- Si el código de las tareas parte del mismo esquema, tendremos que sugerir cambios de mapping de las tareas, o bien redistribuciones de carga de la operación (si los tamaños de mensajes no fueran iguales).

- En casos de retraso del receptor, intentar adelantar en el tiempo examinando dependencias, o mapping, o por contra si las tareas emisoras pueden retrasarse adelantando cómputo.

4.10.3.4. Colectivas N a N

En este tipo de colectivas, cada tarea participante es a su vez emisora y receptora de mensajes. Debido a esta interrelación de las tareas y de los mensajes que hay que componer para realizar el envío y recepción posterior, la aparición tardía de una o más tareas frena el inició global. Así como la aparición de la operación en las tareas en una gran dispersión de tiempos, causa diversos tiempos de espera a algunas, o a todas las tareas.

En particular aquellas tareas que se alejan del tiempo medio de aparición sufren grandes retrasos en el inicio de la operación. En particular podemos considerar, bien el tiempo de espera para cada tarea como la diferencia de entrada de su operación con respecto el tiempo de la ultima en entrar. O tener en cuenta el tiempo perdido global como la suma de los perdidos en cada tarea, de forma que así podemos utilizar este índice para comparar la importancia con otro instancia de este problema, en la etapa de clasificación.

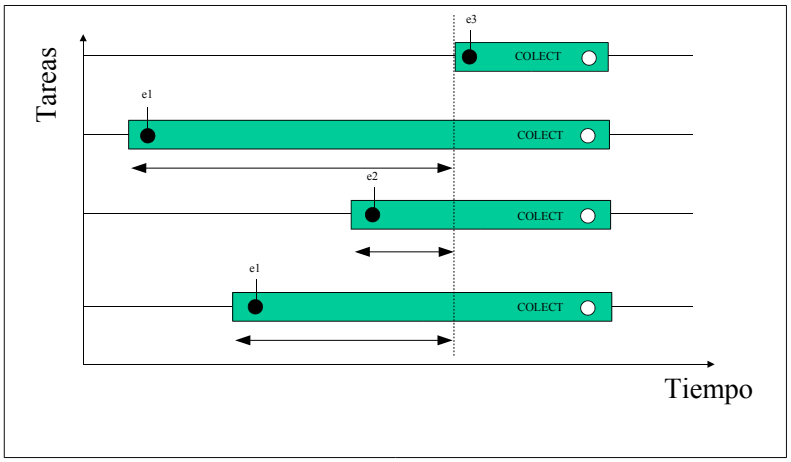

Figura 4.15: Ineficiencia causada por colectivas N a N

Los casos observados en este problema, variaran básicamente debido a las distribuciones que se produzcan en los tiempos de entrada, a su vez que los de salida. Ya que este tipo de problemas N a N aparecen debido a la existencia de las tres fases por las que pasan las operaciones. Teniendo en cuenta: 1) Repartimiento inicial; 2) Formación de los datos; 3) Envío final a todas las tareas. Dependiendo el desarrollo de las fases de los diferentes algoritmos de implementación de bajo nivel de las primitivas. Por el análisis de traza, podemos básicamente solo observar los eventos de entrada a las tareas, definiendo la distribución de entrada, y las a salidas de la operación colectiva. En particular observaremos la causa de retraso de inicio por la aparición de la ultima tarea (o del grupo de ellas que se retarde) dentro de la operación, y la salida de la operación, que nos puede indicar, en el caso de N a N con diferentes tamaños de datos, aquellas tareas que tarden más en completar la operación.

4.10.4. Estructurales

Como nivel adicional de ineficiencias, mostramos aquí algunos análisis que ya estaban disponibles en KappaPI [Esp00, Esp00a], y otros añadidos (caso del pipeline), para la determinación de estructuras de colaboración entre las diferentes tareas de la aplicación (otros trabajos similares [Hub99, Hub01]).

No estamos en este caso hablando de una estructura de patrón de ineficiencia, sino de una fase de ejecución que sigue un patrón de un paradigma de progra-

mación, en el cual examinamos sus parámetros de funcionamiento para detectar ineficiencias en su uso.

Estos análisis tienen la función de determinar, si fases particulares de la ejecución de la aplicación (o toda ella), poseen un determinado paradigma de programación (como los mostrados en el capítulo 1), y permitir bajo este conocimiento, un análisis más detallado de los factores que intervienen, teniendo en cuenta pequeños modelos del paradigma, poder determinar que parámetros están causando ineficiencias en el paradigma usado, y como estas pueden relacionarse finalmente con el código fuente que las genero, proponiendo sugerencias de mejora de los parámetros de funcionamiento que use el paradigma.

4.10.4.1. Master worker

En la detección de este paradigma hay que tener en cuenta algunas características del entorno de paso de mensajes. Ya que normalmente en la creación en PVM, suele realizarse por medio de aprovechar la creación dinámica de tareas mediante la primitiva *pvm_spawn*, la cual permite iniciar las tareas esclavas desde la tarea master (este era el tipo de detección usado en KappaPI [Esp00, Esp00b]).

En el caso de MPI, este paradigma se emplea de forma diferente, debido a los grupos estáticos de procesos, las tareas son iniciadas generalmente con un modelo SPMD, con copias de la misma tarea, la cual dependiendo de su rango dentro del comunicador MPI utilizado, elige comportarse como tarea master o tareas esclavas. Aunque el mismo modelo también puede implementarse, como tareas MIMD donde se especifique su inicio por fichero de configuración al lanzar la aplicación MPI. Este ultimo caso es dependiente de la implementación MPI, y su entorno de ejecución, siendo más habitual la primera opción.

Por esta razón la detección de estructuras de tipo Master-Slave no se realiza por activación de las tareas, sino por el patrón de comunicaciones presente. Básicamente en un estructura Master-Worker, se plantea que una tarea inicia un particionamiento de datos, repartiendo parte de estos datos en forma de mensajes a sus tareas esclavas, estas reciben, tienen un periodo de cómputo, y reenvían sus resultados parciales a la tarea Master, que recoge los resultados. Esto en un modelo simple de Master-Slave, en otros más complejos, pueden aparecer iteraciones de este modelo, o con variaciones de la carga ofrecida a los slaves en cada iteración mediante mecanismos de balanceo de carga dependiendo del estado de las tareas y su rendimiento

En el caso de los paradigmas de programación Master-Worker hay que tener en cuenta varios aspectos a la hora de considerar sus prestaciones:

- Granularidad de las tareas: relación entre los bytes recibidos (y/o enviados) por la tarea, y el cómputo realizado. Una medida práctica del ratio entre computación y comunicaciones de las tareas. Se puede realizar un estudio previo, con algunos cálculos del modelo computacional, y la anchura de banda para tal de tener una estimación superior de los niveles de granuralidad alcanzables.

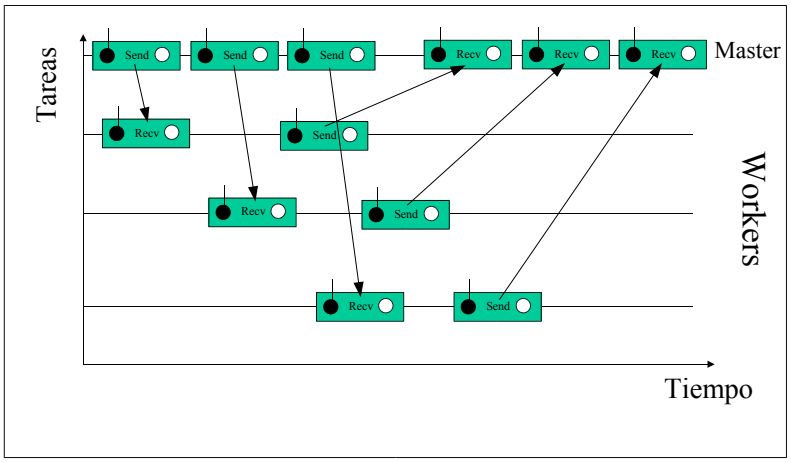

Figura 4.16: Estructura en Master Worker

- Organización de las comunicaciones: Básicamente decidir número de mensajes frente a tamaño de los mensajes (su longitud). Mediante el uso mensajes largos se reduce el tiempo de arranque (influido por la latencia), pero no comporta necesariamente que se reduzca el tiempo global, dependiendo de posibles congestiones de la red, así como de que las tareas se encuentren preparadas para procesar los mensajes. Hay casos en que los mensajes pequeños pueden estar solapados con la computación, y el overhead es enmascarado. Normalmente tanto la habilidad de solapar comunicaciones y cómputo, como el número de mensajes óptimo a enviar, son muy dependientes de la aplicación (y de estudio obligado si la queremos optimizar). Aún así existen diversas técnicas para obtener, como veremos, estimaciones dependiendo de modelos de comportamiento de Master-Worker, y de la observación del histórico del comportamiento de los procesos.

- También es relevante en modelos más complejos de MW, el tipo de paralelismo utilizado en la aplicación. Si solo se realiza particionado de datos o también hay algún tipo de paralelismo funcional. En este caso nos pueden afectar si las tareas worker efectúan exactamente todas el mismo cómputo, o por si contra hay variaciones (granularidad de cómputo) dependiendo de los datos (por ejemplo por uso de diferentes algoritmos). Por otro lado también puede afectar que la tarea Master tenga además del envió y recolección de datos, que efectuar otras tareas de computación, que podrían frenar el inicio

de una repartición de datos cuando ya hay workers preparados, o podrían frenar la recolección cuando los workers ya tienen los datos preparados para enviar.

- En entornos heterogéneos [Alm03, Ban04], las capacidades computacionales diferentes de cada máquina, nos puede obligar a que la partición del problema se vea condicionada por la configuración de las máquinas (o el estado de carga de estas).

- Variación de la latencia de los mensajes: La alta latencia de los mensajes puede tener razones múltiples, por la distancia existente entre emisor y receptor, la congestión de la red, ya sea por mensajes de la propia aplicación o por otras aplicaciones coexistentes (pertenecientes a otros usuarios o al propio sistema) durante la ejecución.

- Por otro lado hay toda una serie de características de naturaleza dinámica dependiendo del entorno que pueden variar durante la ejecución, como el ejemplo anterior de la latencia, las prestaciones computacionales, la anchura de banda, que dependerán de la compartición y uso dedicado o no del sistema a la aplicación paralela. Esto puede comportar problemas en la optimización de prestaciones debido a las condiciones cambiantes en cada ejecución. Si se producen estas variaciones, sera necesario algún uso de mecanismos de balanceo de carga en las aplicaciones MW, que se traducirá en diferentes ratios de computación/tamaño mensajes para los workers, o en la propia elección de estos (en mapping estático o dinámico de la aplicación paralela).

En este caso, durante el análisis del MW podemos detectar a partir de la observación de los eventos, diferentes características del paradigma. En particular debido al uso de las comunicaciones punto a punto, y la capacidad de capturar durante la monitorización, los tamaños de mensajes utilizados, así como las diferentes iteraciones que sucedan de la misma fase de MW podemos deducir factores como la frecuencia de generación de mensajes, determinar que carga suponen, y analizar los posibles desbalanceos o retardos que surjan.

Estos análisis nos permitirán por ejemplo, determinar el número de workers que podemos considerar óptimos, para la carga de trabajo, y evitar así tiempos de bloqueo altos en la tarea master. O determinar posibles agrupamientos de mensajes, si estos no son suficientes para mantener unos ratios adecuados de la relación cómputo prestaciones dentro del paradigma.

En el caso de la detección de esta estructura, y del análisis de sus parámetros bajo un determinado modelo deberíamos determinar las ineficiencias causadas debido a los casos de:

- Desbalanceo de carga, donde las etapas de computación entre recepción y emisión de resultados producen acusables diferencias entre las tareas worker. Se pueden observar diferencias entre las tareas (mismo o diferente código) si el tamaño de mensajes es el mismo, o bien sugerir diferentes mappings.

- Número incorrecto de esclavos, bien sea por pocos, causando tiempo elevado de espera en la tarea de Master, o bien por demasiados causando una baja ratio entre cómputo y comunicaciones en los workers.

- Retardos en las comunicaciones de algunos procesos, debido o bien a problemas de balanceo previos, o bien a repartición no uniforme de datos. Así como a posibles problemas de ancho de banda, latencia, o contención de los mensajes. Otra causa de comunicaciones puede ser debida a efectos de carrera en el Master por envío de los datos iniciales, o bien por recepción en carrera desde los workers. Debido a un orden incorrecto en envío respecto a las tareas worker preparadas para comenzar, o bien orden incorrecto de recepción respecto a las tareas worker con resultados disponibles.

- Relación incorrecta en los mensajes del ratio entre tamaño de estos respecto al número de envíos. En este caso si se presentan diferentes fases iterativas del modelo master-worker, podemos evaluar los ratios y decidir agrupar mensajes, o por el contrario desagruparlos en función del tamaño y número de estos.

4.10.4.2. Pipeline

En las estructuras en Pipeline, se producen repeticiones de comportamiento, mediante iteraciones en las que se produce la llegada de un mensaje, el proceso de los datos para producir una salida que sera enviada a la tarea vecina. Habitualmente todas las tareas que forman parte de la cadena, procesan de la misma manera, aunque la funcionalidad de calculo de cada nodo puede ser diferente.

En esta estructura es importante mantener un flujo constante de información, que permita que las etapas (nodos con entrada y salida) permitan procesar adecuadamente los datos de entrada, y produzcan la salida en un tiempo razonable, de manera que las tareas consumidoras siguientes no se vean frenadas en su avance por el retraso de las etapa anterior.

En el pipeline, son parámetros importantes el número de etapas que lo forman, y el tiempo de ciclo, o tiempo de proceso de la etapa más larga. El tiempo de ciclo, influye , dependiendo si se utiliza un modo síncrono o asíncrono de avance en el pipeline, si nos encontrásemos en arquitecturas de sistema síncronas (tipo SIMD), el tiempo de ciclo nos definiría el avance entre etapas. En caso contrario, por ejemplo en una arquitectura MIMD o en cluster, el tiempo de ciclo, nos influirá habitualmente en el tamaño de la cola de mensajes de entrada o salida de las etapas, que se verán incrementadas dependiendo de las relaciones de tiempo de etapa entre las etapas vecinas, así como su ratio global respecto al tiempo de ciclo de la etapa más lenta.

En un modelo puro de pipeline lineal, la unidad de cómputo que se encomienda al pipeline, se ve completada cuando ha pasado por todas las fases y produce el resultado final. En este modelo podemos examinar las prestaciones con la métrica

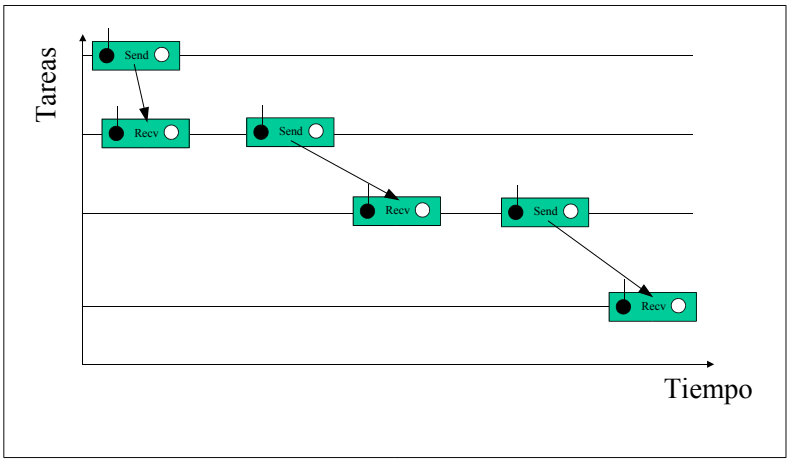

Figura 4.17: Estructura en Pipeline

de throughput, como medida de unidades de cómputo completadas por unidad de tiempo.

Dependiendo de los casos podemos sugerir una serie de mejoras a realizar en el pipeline:

- Si hay una etapa claramente consumidora de tiempo, que crea un cuello de botella debido a su tiempo de ciclo, podemos realizar dos posibilidades: a) Dotar al sistema de varias copias (dependiendo de la carga, y los tiempos relativos de las otras tareas) que puedan crear caminos alternativos para los mensajes desde la tarea previa en el pipe. Esto sugiere replicar la tarea problemática con varias iguales, de manera que se tendrá que cambiar en código el envió de la tarea precedente a las diferentes copias existentes de la tarea, por ejemplo usando un envío rotatorio de tareas destino. Y en la tarea adyacente a la replicada, será necesario captar mensajes de forma también rotatoria desde las copias precedentes. Podrían usarse variaciones de este patrón si se apreciaran diferentes niveles de carga desde las tareas. En este caso la recomendación hacia el usuario, se hace requiriendo su actuación sobre las comunicaciones de las tareas precedentes y posteriores. Así mismo será necesaria la creación de tareas adicionales de forma dinámica, o bien por medio de estáticas en el momento de inicio de la aplicación.

- otra posibilidad b), es la partición de la tarea en algunas más simples, que equivaldría a una subdivisión de la funcionalidad, siempre que esta sea po-

sible. La idea es crear subtareas que sean más parecidas en tiempo al resto de tareas y que permita trabajar con tiempos de ciclo menores. En este caso dependeremos fuertemente de la granularidad de las particiones que podamos efectuar, y los tiempos extra que introduzcamos como resultado de las comunicaciones adicionales, que dependerán del ratio de comunicaciones cómputo de la tarea, y las nuevas particiones de ella. Esta recomendación de subdivisión, no seria práctica de cara al usuario, porque solo puede informarsele de donde se encuentra el problema, pero es difícil encontrar un esquema de sugerencias para realizar las particiones.

- Otro aspecto es la relación de tiempo de cómputo de cada etapa, con respecto a los tiempos de comunicación (tanto en la fase de entrada como en la fase de salida). Esta relación tiene que ser lo suficientemente grande para que el cómputo sea mucho mayor que el tiempo de comunicaciones. Si esto no sucede podemos pensar en fusionar etapas de pipeline, para que tengan un tiempo mayor de cómputo, a expensas de disminuir la comunicación por red.

- Otra posible optimización esta en la agregación de mensajes que podría realizar una tarea, para pasarla a la siguiente en tamaños mayores, siempre que la funcionalidad lo permitiese.

4.11. Relación de los problemas con el código fuente

En el análisis de casos, necesitamos establecer el valor de condiciones, o demandas adicionales de información mediante determinadas peticiones de análisis estático del código fuente de la aplicación.

Partimos de cada problema junto con sus eventos, información temporal y localizaciones en el código incluidas. Permitiendo así conocer para cada evento que forma parte del problema su posición estática dentro del código de la aplicación, en forma de número de linea, y fichero de código fuente donde se ha producido, además de la información dinámica de tarea y máquina donde se produjo. Además en muchos de los casos, como hemos visto, serán necesarios análisis adicionales sobre los parámetros que intervienen directamente en las llamadas, o los que puedan deducirse de la información dinámica incluida en los eventos, mediante procesamientos posteriores.

En cada uno de los casos de análisis del código fuente tenemos que testear condiciones simples, como dependencias de datos en los parámetros, dependencias entre variables, tipos de algunos parámetros, etc ... Estos análisis los implementamos con una pequeña API de análisis estático de código fuente, que hemos denominado quickParsers API, de manera que definimos un conjunto básico de funciones de análisis, con una tarea clara determinada, en la búsqueda de hipótesis o condiciones en el código fuente.

Esta API, es un compromiso entre complejidad de los procesos de análisis y búsqueda en el código fuente, y la utilidad de la información que se espera obtener.

En la literatura existen complejos entornos de análisis de código, que consideramos sobredimensionados para el análisis que necesitamos. Pero es un punto interesante para el trabajo futuro poder extender los tratamientos de análisis de código fuente a un nivel superior, si realmente nos proporcionaran una mejor aproximación a la realidad en las recomendaciones que emitamos hacia el usuario final.

En nuestro caso complementamos el uso de la API de análisis, para proporcionar una visión dual del código, mediante la incorporación de una segunda forma de representación, basada en la representación de ambientes de código (aportando análisis de bloques). En este caso usando la representación de código fuente C (en nuestro caso) basada en representación XML, permitiendo realizar así consultas sobre la estructura del código fuente, y las posiciones de funciones llamadas dentro de otra función así como las posiciones de llamada.

Así un análisis del código fuente podrá producirse en dos fases, una por representación del código en SIR, relacionada con el ambiente del código (bloques, condicionales, bucles, etc.), que nos permite identificar la estructura del código en los puntos donde se produzcan las llamadas a sistemas. Y otra fase, parseo de código, donde se puedan verificar las relaciones locales del código, en función de dependencias, parámetros de llamadas, etc. Estas dos fases se complementan entre ellas para proporcionarnos la información sobre el código para los casos de uso de los problemas.

El análisis final de código puede verse como un análisis multidimensional, ya que nos encontramos con las fases, de por una parte: 1) Representación de los elementos y estructura del código, realizada en SIR; 2) Examen directo del código fuente estático detallado; y 3) Complementada con la información dinámica obtenida de la traza de la ejecución de la aplicación, que nos aporta posiciones de código donde se han producido llamadas de paso de mensajes, y en particular información colateral que se puede deducir juntando con 1) y 2) a propósito de condicionales y bucles, para conocer si en un determinado punto se han o no activado, o en que estado de desarrollo (que iteración, condición) se encontraban.

Este análisis tridimensional es el que nos permite extraer información que complemente el análisis de los problemas, determinando posiblemente en que casos de uso nos encontramos, y dando la información suficiente para poder emitir sugerencias aplicables al código, y comprensibles en la sintaxis y semántica usada por el desarrollador, a la hora de realizar su aplicación.

4.11.1. Parseo rápido de código

En nuestro esquema, tenemos que tener en cuenta que gracias a la visión complementaria de la estructura del código (que expondremos en el apartado dedicado a SIR), podemos conocer el ambiente de bloque de un determinado evento, así como las zonas de funciones donde esta presente, o que producen las llamadas a el.

En el caso de análisis de código, hay que pensar que con la información dinámica complementamos la información estática del código, con lo que ha sucedido en la aplicación, un caso particular claro de esto, son las construcciones condicionales,

y los bucles.

En este caso particular cuando analizamos el ambiente donde se ha producido una determinada llamada, que tenemos registrada en los eventos, sabemos que se ha producido realmente. Y conocemos por tanto que el ambiente ha estado ejecutado, e incluso podemos disponer de datos extra debido a que los parámetros (que disponemos) de la primitiva representen y/o aporten datos extra sobre el ambiente.

El caso típico es la información de condicionales if-then, conoceremos el valor booleano de la condición, y sabremos que se ha producido, o bien en casos while y do-while sabemos que la condición se abra cumplido almenos una vez, o incluso podemos disponer de las veces que se haya cumplido si ha aparecido en varios problemas o en repeticiones de estos.

En casos de bucles basados en repeticiones (for) o en los mismos condicionales (while, do-while) si los parámetros de la llamada intervienen en ellos, podemos conocer datos sobre cuando se cumplirá la condición, o porque se ha cumplido en un determinado momento.

4.11.1.1. Análisis de dependencias

El análisis de dependencias [Psa04], es un campo que se ha desarrollado básicamente en el aérea de compiladores, y en especial en los compiladores paralelizantes (para paralelismo implícito) en códigos Fortran o HPF, para paradigmas de memoria compartida.

El concepto de dependencia de datos, se refiere a una relación de orden parcial entre las sentencias de un programa. En particular, dos sentencias son dependientes en datos si ambas acceden a la misma posición de memoria, y al menos una de ellas escribe en ella. En la presencia de una dependencia de datos, el orden de ejecución en el programa original ha de preservarse. Si las sentencias no exhiben dependencias de datos pueden ejecutarse en cualquier orden, y en particular de forma paralela.

En variables escalares este análisis puede realizarse mediante análisis del flujo del programa, y determinar cuando son usadas y con que operaciones de lectura o escritura. Para variables de tipo array, es necesario analizar los índices utilizados, para determinar si una o más operaciones actúan sobre los mismos datos, o sobre conjuntos parcialmente solapados. En particular se presenta el problema de detectar múltiples referencias a los arrays, y a sus posiciones, tanto por otros array como por variables escalares. En casos multidimensionales, llega a ser complejo, ya que se plantean sistemas de ecuaciones para dilucidar si realmente los índices hacen referencia a un acceso a una misma posición o no.

También pueden examinarse dependencias en bucles, por si dos sentencias en cada instancia del bucle son independientes o no entre ejecuciones iterativas bucle, o si estas sentencias son independientes dentro del bucle. Si son independientes respecto las iteraciones del bucle, entonces tenemos posibilidades de paralelizar la realización del bucle.

Normalmente las dependencias de datos se analizan teniendo en cuenta los

denominados vectores de distancias o dirección. Cada sentencia puede especificarse, con sus dependencias hacia otras situadas a cierta medida de localización (por ejemplo número de sentencias intermedias, o número de iteraciones de bucle), o si la dependencia es hacia atras o adelante en la ejecución.

Para el caso de las dependencias en las aplicaciones por paso de mensajes, tenemos que tener en cuenta que los eventos tratados relacionados con las primitivas de comunicación disponen de una serie de parámetros relacionados, con los datos que van a ser enviados o recibidos, como mensaje.

Estos datos en forma de parámetros, en ocasiones, deberemos verificar su no dependencia respecto otros datos presentes, ya sea en el ambiente local de realización de la llamada a la primitiva, en el ambiente local de la función (o procedimiento), o en ambientes superiores a nivel módulo o global.

En ocasiones cuando exista dependencia, deberemos comprobar el tipo de esta: si se producen simples copias (o referencias), cálculos simples adicionales, o si por contra estamos en aplicación de una función (más o menos compleja) que obtiene unos datos a partir de los otros. Esto nos va afectar en si es posible, su reformulación, para evitar las dependencias, o reescribirlas de una forma más flexible, o en el caso de dependencias complejas, como mínimo ser capaces de indicar los puntos adecuados en las recomendaciones al desarrollador, para que este conozca el contexto más detallado posible de las dependencias existentes.

4.11.1.2. Análisis mediante QuickParser API

Tanto en el propio análisis de los casos, como en la posterior construcción de las recomendaciones finales, necesitaremos examinar el código, en unos puntos concretos que nos permitan extraer la información necesaria. Esta información nos permitirá precisar los casos del análisis del problema, en aquellos que la decisión final dependa del código presente, o las dependencias de datos que se vean involucradas.

En la generación de recomendaciones, será imprescindible incluir información exacta de la posición donde aplicar los cambios, y como ponerla en contexto con las zonas adecuadas de la aplicación que será necesario examinar por parte del desarrollador, ya sean lineas de código fuente, con respecto a llamadas, bloques de código, funciones, módulos o ficheros de código fuente.

Debido a los análisis estáticos de código necesarios, la implementación de este análisis estático, se ha realizado mediante la implementación de una API denominada QuickParsers, cuyo objetivo es la realización de un grupo de análisis concretos sobre el código, que hemos identificado como necesarios, para las prestaciones que esperamos obtener de este análisis.

La API denominada QuickParsers, está formada por una serie de funcionalidades:

- Modelo de código fuente: La aplicación está modelada por una serie de ficheros de código fuente (en lenguaje C para nuestro caso). Este modelo incluye

los ficheros, funciones que se implementan en estos ficheros, y ambientes (bloques de código). Toda esta información es de tipo dinámico durante el análisis. No hay un análisis total del código previo, sino que la información es añadida a medida, que el analísis se desarrollada de forma incremental.

- API de gestión de código básica: Nos permite implementar sencillas operaciones, que nos permiten obtener información básica sobre el código, que normalmente suele añadirse posteriormente al modelo del código. Funciones como localizar una determinada llamada en el código, determinar su posición concreta en el código (en forma de número de lineas dentro de el), situar el principio y final del bloque de código que contiene una determinada llamada, etc...

- API de análisis propiamente: Implementamos (de forma abierta) todas aquellas funcionalidades que hemos identificado como necesarias durante el análisis de casos. Respecto a diferentes ámbitos de análisis necesarios: a) Determinar si una determinada llamada o región de código es movible hacia zonas más adelante o atras en el código; b) Determinar condiciones que favorecen o impiden la producción de una llamada, como condiciones de bucles o condicionales tipo if, while, do ...; c) Dependencias entre un dato determinado dentro de su bloque, función o ámbito global, o bien dependencias entre dos llamadas o zonas de código. Así como análisis del tipo de dependencias presentes; d) Determinar el tipo de zona (o bloque) de código donde se produce la llamada, así como la determinación de las condiciones que la permiten.

Esta API nos permite disponer de la capacidad de completar la determinación de los datos necesarios para complementar el análisis de los casos de los problemas, y también como veremos obtener la información extra sobre el código que sea necesaria, para obtener recomendaciones finales más precisas.

La API nos implementa uno de los ejes dimensionales del análisis del código, que complementa la información obtenida en los eventos de los problemas de ineficiencia detectados (información dinámica obtenida desde la ejecución de la aplicación). Y a su vez complementaremos con un tercer eje de representación del código, que nos permitirá poner en correspondencia los diferentes niveles de información sobre el código fuente, ya sea desde el eje dinámico de ejecución, el análisis local preciso, y el análisis estructural que aportara esta representación intermedia del código mediante el uso de la especificación SIR.

4.11.2. SIR: búsqueda de ámbito de bloque

La especificación SIR (desarrollada en el proyecto APART) nos permite obtener una representación abstracta del código fuente de programas procedurales y orientados a objeto.

Uno de los objetivos del diseño de SIR, es evitar que las herramientas de análisis de prestaciones, necesiten construir diferentes componentes para diferentes len-

guajes de programación, una tarea tediosa y con un esfuerzo grande en el tiempo.

Así la introducción de SIR nos permitirá poder aportar información del código fuente de las aplicaciones a la herramienta de análisis, y poder procesarla mediante una cierta API con métodos comunes de acceso a la representación de las diferentes entidades del código fuente. El objetivo final de estos trabajos en SIR es proporcionar una interfaz de alto nivel y portable para facilitar la implementación de herramientas de instrumentación, profiling y monitorización, basándolas en la representación SIR que permite abstraer diferentes lenguajes. La representación SIR también tiene una parte de especificación utilizable, para describir peticiones de instrumentación del programa, que permiten especificar cuando obtener una determinada métrica, con que frecuencia, y que datos hay que tomar en ella. Así una herramienta, que utilizase estas posibilidades, podría examinar la representación SIR con una interfaz estándar, realizar peticiones para obtener información, y permitir especificar instrumentación para regiones especificas de código.

Las especificaciones de SIR, se han realizado basadas en XML, lo que facilita posteriormente el uso de APIs comunes de XML para procesar y lanzar acciones dependiendo de las entidades presentes en la especificación. Otro uso, es la descripción de estados de la aplicación en ejecución, pudiendo describir la situación de nodos, procesos, y comunicaciones que se están siendo usadas en un determinado momento.

Para la especificación de código en SIR son elementos característicos de la representación SIR:

- SIR, entidad que representa toda la especificación del programa.
- unit,representado diferentes entidades de código, ya sea módulos, subrutinas, funciones, programa principal en un lenguaje procedural, o bien clases y métodos en un orientado a objetos.
- codeRegion, los diferentes posibles bloques de construcción del programa, como bloques, asignaciones, bucles, condicionales, casos de test, bucles condicionales, etc. En particular, puede disponer de un atributo de posición donde se define su localización física en el código.
- callee, describiendo un punto de llamada
- expresión, habitualmente describiendo una evaluación que se tiene que realizar como condición de condional o control de bucle.
- position expresando una localización en el código, en forma de números de lineas y columnas en el código.

Veamos un ejemplo, suponiendo el siguiente código (en lenguaje C), observamos una construcción condicional de tipo if (una coderegion) incluyendo el uso de bloques (otra coderegion), caller, y expresiones:

teniendo en cuenta el código original C:

Código original

```
1  if (f(n) > g(m)) {
2    a = 10;
3  } else {
4    flag = false;
5  }
```

obtenemos una representación SIR, con las construcciones relacionadas:

Representación SIR

```
1  <!-- i(n) es cada identificador de elemento -->
2  <codeRegion type="if" id="i1">
3    <!-- if (...) -->
4    <codeRegion type="block" id="i2">
5      <expression>
6        <codeRegion type="call" id="i3">
7          <!-- f(n) -->
8          <callee id="f"/>
9        </codeRegion>
10       <codeRegion type="call" id="i4">
11         <!-- g(m) -->
12         <callee id="g"/>
13       </codeRegion>
14     </expression>
15     <codeRegion type="assignment" id="i5"/>
16       <!-- a = 10 -->
17     </codeRegion>
18   <codeRegion type="block" id="i6"> <!-- else -->
19     <codeRegion type="assignment" id="i7"/>
20             <!-- flag = false -->
21     </codeRegion>
22   </codeRegion>
23 </codeRegion>
```

Este trozo de programa es mapeado a SIR como la siguiente representación SIR, que permite su exposición como una representación XML, que se puede manipular para localizar determinados puntos en el código, o para poner en contexto estructural (dentro del código general) alguna de las partes examinadas. En este caso la representación queda como:

donde observamos el anidamiento de las coderegion representando la estructura jerárquica del código. También en particular observar la capacidad de detectar los puntos de llamada, y los parámetros usados.

En KappaPI2 la representación usada con SIR, nos permite aportar análisis de bloques de código. Permitiendo detectar la estructura de una región de código con respecto a las llamadas de paso de mensaje que se produzcan, y relacionándolas con los parámetros de los condicionales o bucles donde se produzcan.

En nuestra herramienta, hemos desarrollado un parser, que nos permite el volcado de programas C a la representación SIR. Permitiendo identificar las llamadas de paso de mensajes, examinando su posición, y obteniendo una jerarquización de

los bloques donde se encuentran, pudiendo identificar la estructura. También hemos desarrollado una pequeña API que nos permite examinar estas condiciones en el código, y a su vez nos permite extraer información (sobre posición, estructura) que podamos después utilizar para un análisis estático detallado del código.

4.11.3. Elección de casos de uso

Para la elección final de los casos de uso, tenemos que observar las condiciones de código que se hayan cumplido, así como si algunas de las reestructuraciones de código que sugeriremos realizar, son posibles dentro del caso.

Normalmente usaremos una estructura if-then-else de casos de uso, para establecer prioridades entre ellos, pero podríamos decidir si existiese una o más de ellas proporcionar las dos o más soluciones posibles para que el usuario evaluase las actuaciones que le podamos proponer.

En este sentido podría ser interesante utilizar, en trabajo futuro, posibilidades de evaluación, de los cambios propuestos, mediante el estudio de las mejoras esperadas que introduciríamos. Pudiendo tener unas medidas heurísticas de la calidad de mejora posible de los casos de uso detectados. Aunque también con un análisis más preciso podríamos reducir el número de casos de uso que podrían darse, debido a que tendríamos en principio información incompleta que puede no darnos capacidad de decisión suficiente para escoger cual de los casos es el correcto.

4.12. Emisión de sugerencias

Una vez determinados los casos de uso del problema, procedemos a una emisión de sugerencias hacia el usuario. Esta emisión final de sugerencias (*hints*) al usuario final, la hemos realizado con vistas a proporcionar un retorno directo al usuario para mejorar su aplicación, y aumentar las prestaciones que podrá obtener.

Detectado un caso de uso del problema, y realizadas las comprobaciones pertinentes, en forma de análisis del código fuente, daremos información directa de actuación sobre el código de la aplicación.

En la especificación abierta del análisis de los casos de uso, podemos complementar estos con la especificación de una serie de *templates* que darán el mensaje final de cara al usuario.

Cada problema nos presenta una serie de casos de uso, junto con el template que usaremos de recomendación final. Este template es el mensaje que recibirá el usuario teniendo en cuenta, todos aquellos datos que obtenidos del análisis multidimensional del código fuente permita identificar de forma precisa la zona (o zonas) donde deberemos actuar sobre el código de nuestra aplicación, así como aquellos movimientos de código o reestructuraciones que serán necesarias para evitar o disminuir los efectos del problema causante de la ineficiencia.

4.13. Conclusiones

La arquitectura flexible de fases de detección, clasificación, análisis de causas, y relación con el código fuente para emisión final de sugerencias, nos proporciona una arquitecuta de construcción de herramientas en el que hemos podido desarrollar en forma de prueba de concepto el prototipo de la herramienta KappaPI 2.

La arquitectura en que se basa la herramienta KappaPI 2, posee dos niveles de aportación de conocimiento, que la hace adaptable a cualquier base de conocimiento actual, o que pueda surgir, sobre patrones de ineficiencias en aplicaciones de paso de mensajes. Permitiéndonos aportar conocimiento de la estructura del problema, y del análisis de sus casos de uso, para determinar que causas han podido ocurrir para provocar la aparición del problema.

El uso del árbol de decisión nos permite una clasificación del conocimiento incorporado, de manera que añadimos conocimiento en principio no presente en la especificación inicial, sobre solapamientos, o composiciones de problemas, por otra parte difícilmente detectables cuando podemos encontrarnos con problemas con estructuras definidas grandes o solapadas.

Así en este trabajo hemos disertado sobre una nueva herramienta de análisis automático de prestaciones, orientada al usuario final para facilitar la mejora de prestaciones de las aplicaciones escritas en un modelo de paso de mensajes, evitando que el usuario tenga que disponer de altos grados de conocimiento para interpretar los resultados del análisis y pueda incorporar las mejoras propuestas en la aplicación.

Nuestra herramienta KappaPI 2, está basada en la arquitectura abierta y flexible, que permite incorporar nuevos modelos de problemas de prestaciones y conocimiento sobre su análisis. Y es capaz de realizar sugerencias sobre el código fuente para mejorar el tiempo de ejecución de la aplicación, y evitando (y/o minimizando) los problemas de prestaciones detectados.

Capítulo 5

Casos de estudio: Experimentación

En este capítulo examinamos algunos casos de estudio donde se ha aplicado la herramienta KappaPI 2, de manera que hemos podido validar las diferentes fases del análisis de prestaciones, propuestas en la arquitectura, y comprobar la bondad de los resultados obtenidos en diferentes ámbitos de aplicación de la herramienta prototipo de la arquitectura.

Con el fin de validar la arquitectura sobre la que se ha implementado la herramienta automática de análisis de prestaciones KappaPI 2, hemos desarrollado diferentes conjuntos de experimentos para validar las diferentes fases de la arquitectura de KappaPI 2. Así como varios experimentos para examinar la usabilidad, y rentabilidad de la herramienta sobre aplicaciones reales.

El conjunto de tests, está compuesto de un conjunto de aplicaciones de paso de mensajes escritas en lenguaje C para entornos PVM y MPI. Hemos utilizado algunos benchmarks sintéticos creados para detectar un patrón de problema particular de prestaciones, o bien repeticiones del mismo problema, o combinaciones simples de varios problemas. Estos benchmarks se han creado usando el toolkit APART testsuite [Moh03, Ger04], que ofrece un entorno flexible de creación de benchmarks sintéticos sobre MPI. Otros benchmarks usados son benchmarks paralelos clásicos, disponibles en versión en lenguaje C para MPI o PVM. Finalmente se han usado varios ejemplos de aplicaciones paralelas científicas reales, de ámbitos variados, para comprobar la usabilidad, y la escalabilidad de la herramienta bajo diferentes ejecuciones.

Respecto al entorno de pruebas, todos los experimentos se han desarrollado, sobre un cluster de 32 nodos de cómputo, con las siguientes características:

- Un procesador P4 1,8Ghz por nodo.
- 512MB de memoria principal, y 40GB de disco duro.
- Conexiones red mediante FastEthernet (100Mbps)

- El cluster dispone de dos nodos dedicados, portal de entrada, con servicios NIS, Gateway y Firewall. Y un nodo de almacenamiento NFS.

El sistema operativo es GNU/Linux SuSe 8 con kernel 2.4.18, y se dispone de sistemas de paso de mensajes PVM 3.3.11 y MPICH 1.2.5.

Respecto al prototipo de KappaPI 2, es una herramienta desarrollada en Java (compatible j2se 1.4 y 1.5), de aproximadamente 12000 lineas de código fuente (en el estado de prototipo actual), con uso intenso de estructuras de datos proporcionadas por las Java Collections (JFC), y librerias de XML.

5.1. Detección sobre benchmarks sintéticos: APART test-suite

En muchos casos de análisis de prestaciones en otros campos, es habitual aplicar técnicas de uso de benchmarks industriales para el estudio de las prestaciones. Normalmente estos son una serie de códigos de aplicaciones o aplicaciones sintéticas que han estado escritas por usuarios para sus propósitos, que son adoptados por otros, y finalmente propuestas como tests para medir las prestaciones. Esto es importante, porque permite que sean los usuarios (y/o consorcios industriales/educacionales) que definan estos test, y no los propios fabricantes del producto estudiado.

Una de las particularidades de estos tests es que pueden medir desde un simple aspecto, de forma reproducible, hasta todo un conjunto variopinto de casos no identificable a priori, o incluso contradictorio entre las posibles optimizaciones de prestaciones a realizar. En muchos casos puede no llegar a estar claro, por el gran numero de factores, que es lo que indica exactamente los resultados de un determinado benchmark de una maquina o plataforma software.

En el caso de paso de mensajes, hasta la aparición de estandars como PVM y MPI, existían muchas implementaciones diferentes de entornos de mensajes, normalmente no muy portables, y fuertemente adaptadas a las arquitecturas de los sistemas paralelos de cada fabricante, que solia proporcionar sus propios benchmarks. Era prácticamente imposible poder comparar las máquinas, hasta la aparición de benchmarks tales como los usados por ejemplo en el TOP500 [Top05], en los cuáles se intenta medir la capacidad bruta de computo paralelo de las máquinas de forma independiente a sus arquitecturas.

Con el advenimiento de entornos de paso de mensajes estandard como PVM, y MPI, han surgido diferentes aplicaciones que se han usado para medir las prestaciones [Hey00], normalmente en forma de kernels como por ejemplo los kernel NAS, o como porciones de código típicas extraídas de aplicaciones paralelas de diversos campos científicos. Aunque aún así hay dificultades por los lenguajes utilizados, muchos de los benchmarks son código Fortran, hay poca disponibilidad de benchmarks en C, y muchos fabricantes tienen especialmente en los compiladores capacidades para realizar una serie de optimizaciones especificas para sus arqui-

tecturas. La elección de los kernels no es fácil, tiene que evitarse precisamente que existan overheads en ellos que permitan crear optimizaciones a nivel compilador.

Otra aproximación tomada, en el proyecto APART, es la creación de benchmarks ya no para la medida de prestaciones, sino para la validación de las herramientas de prestaciones. Ya que estas herramientas presentan diversas complejidades adicionales: a) Hay que asegurar que las herramientas de análisis no creen una perturbación elevada, alterando la semántica y perturbando el comportamiento [Net92] de la aplicación en tiempo de ejecución, en particular puede ser grave en aplicaciones no deterministas [Zhe00], o con uso importante de comunicaciones no bloqueantes, y/o con receptor utilizando *wildcards* en el emisor. b) Se necesita verificar que la información de prestaciones recogida, sea suficiente, y la adecuada para detectar los problemas de prestaciones. c) Se necesita verificar que realicen sus subtareas de forma adecuada y que finalmente den un buen índice de problemas encontrados.

APART test suite [Moh03, Ger04] es un conjunto de funciones MPI y OpenMP que permiten la creación de benchmarks sintéticos, reproduciendo problemas de prestaciones de comunicaciones punto a punto, o colectivas. Con el juego de funciones, pueden construirse aplicaciones que incluyan un problema concreto, o bien una repetición o combinación de problemas, e incluir distribuciones estadísticas de los parámetros de los problemas. De esta manera pueden utilizarse como conjunto de test para determinar la eficacia de una herramienta en la fase de reconocimiento de los problemas, delante de diferentes patrones de aparición de estos.

En cuanto al uso final de APART testsuite, en nuestra herramienta KappaPI 2, hemos modelado problemas concretos, para validar la fase de detección de problemas presentes en la base de conocimiento de KappaPI 2, y su eficacia a la hora de proponer recomendaciones al usuario que permitan la mejora de las prestaciones de la aplicación.

Para ello hemos creado, con la ayuda de las funciones del toolkit APART testsuite, un conjunto de benchmarks sintéticos que reflejan diversos problemas individuales. En concreto benchmarks que hemos denominado mpiLS, mpiLR y mpiBS para los correspondientes problemas relacionados con ineficiencias en las comunicaciones punto a punto (LS = Late Sender, LR = Late Receiver, y BS = Blocked Sender). También hemos incorporado una versión propia de un benchmark para pvmLR, codificada directamente en PVM.

Benchmark Empleado	Problemas encontrados
Apart mpiLS	1 LS
Apart mpiLR	0
Apart mpiBS	1 BS
pvmLR	1 LR

Tabla 5.1: Resultados de detección sobre benchmarks sintéticos

Como podemos comprobar en la anterior tabla (5.1), estos han sido correctamente detectados. En particular, cabe examinar el caso del mpiLR, donde no se

encontró problema de ineficiencia, debido a que en este caso el problema es dependiente de la implementación de MPI, ya que depende su aparición del comportamiento (no definido en el estándar MPI) de la primitiva MPI_Send (y posiblemente de los tamaños de buffers y mensajes utilizados), como ya vimos en los modelos de las primitivas de paso de mensajes. En cambio en la implementación del pvmLR si se detecto el problema debido a la espera hasta que apareciese la recepción del envío.

Con una salida parcial de la herramienta de detección se puede observar que problemas se habían detectado, y como afectaban los principales a la ejecución de la aplicación:

Benchmark	Problemas	Idle time (% total time)
Apart LS (mpi)	1LS	54 %
Apart LR (mpi)	0	0 %
Apart BS (mpi)	1BS	45 %
PvmLR (pvm)	1LR	40 %

Tabla 5.2: Importancia de los problemas en los benchmarks sintéticos

En la tabla 5.2 observamos la importancia del problema detectado, en estos casos simples, con respecto al tiempo total de ejecución de la aplicación.

Para ver un primer ejemplo de actuación de la herramienta KappaPI 2 en sus diferentes fases, podemos examinar el caso del benchmark del Late Sender en MPI.

Primero, la herramienta a detectado este problema, basándose en su especificación [Jor05a], y en la representación como camino dentro del árbol de decisión, en forma de dos eventos con una llamada (MPI en este caso) de recepción como evento root, y su evento relacionado de envío. En el caso del benchmark mpiLS, el Late Sender es el único problema encontrado. En el proceso realizado por la herramienta, dos eventos (en realidad cuatro debido al concepto de entrada y salida de una llamada de paso de mensajes) son obtenidos desde la traza de la aplicación, se correlacionan (mediante aliasing) con los eventos de la descripción (en la especificación del patrón del problema). Se recolecta la información de los eventos, con relación a las tareas, tiempos (donde y cuando se han presentado) y posiciones respecto el código fuente (desde donde surgieron las llamadas). En la tabla final de la fase de detección, encontramos una entrada sobre el problema LS, con los datos anteriores y un computo de tiempo idle relacionado con la tarea que realizaba recepción.

En el análisis de causas del LS, necesitamos, como vimos en el anterior capítulo, observar ciertas relaciones en el código fuente de las tareas. En el caso del envío (llamada send) necesitamos comprobar si es posible mover el código de la llamada más arriba (en el código, y por tanto en el tiempo), evitando posibles dependencias de datos. Otro caso, sera, si no es posible el anterior, mover la llamada de recepción ms abajo, moviendo hacia arriba posibles computaciones (tiempo de CPU) que no estén directamente relacionadas mediante dependencias con la llamada. Si ninguno

de estos dos posibles movimientos son posibles (o no disponemos de información suficiente), podemos sugerir un mapping de tareas diferentes, para proporcionar un mapping que permita a la tarea que realiza el envío una realización de la llamada más rápida (siempre que estemos en un entorno heterogéneo, o homogéneo con diferentes condiciones de carga).

En nuestro ejemplo de código analizado, mpiLS, la herramienta KappaPI 2 detecta un problema de Late Sender, relacionado con dos eventos producidos en las llamadas de MPI correspondientes, en un MPI_Send retardado, y un MPI_Recv que se ha producido antes, relacionándolos con sus posiciones en el código, respecto las lineas (3,10) del fichero fuente (ver extractos del código):

Parte del código Apart LS

```
1   for (n=1;n<=reps;n++) {
2     do_work(2);
3     MPI_Send(message, strlen(message)+1, MPI_CHAR,
4              dest, tag, MPI_COMM_WORLD);
5   }
6 }
7 else {
8   source=1;
9   for (n=1;n<=reps;n++) {
10    MPI_Recv(message, 100, MPI_CHAR, source, tag,
11               MPI_COMM_WORLD, &status);
12    do_work(1);
13  }
14 }
```

En el análisis del código fuente es necesario comprobar si la operación de send puede moverse hacia atras. En la representación en SIR, hemos observado que MPI_Send se encuentra en un bloque de un bucle, después en el análisis, habríamos comprobado que no existe dependencias de datos (do_work() es una función usada en Apart testsuite para simular tiempo de computo de CPU). Como resultado la herramienta ha proporcionado la sugerencia de mover la llamada de envió a la primera linea del bucle (por delante de la llamada do_work()). En este caso, la sugerencia a proporcionado un benefició aproximado del 33 % del tiempo en la ejecución del benchmark. En la siguiente tabla resumimos unas pruebas con el benchmark mpiLS, utilizando diversas repeticiones de la aparición del Late Sender, donde se ha aplicado la recomendación de la herramienta, se presentan los tiempos de ejecución del benchmark original, y del benchmark con la recomendación aplicada.

Repeticiones LS	Tiempo original	Tiempo recomendación
1	15.42 sec	10,48 sec
20	1m 0,6 sec	40,55 sec
50	2m 31 sec	1m 41 sec

Tabla 5.3: Benchmark mpiLS con repeticiones del problema LS

obteniendo el beneficio de la recomendación de aproximadamente 33 % menor tiempo de ejecución del benchmark.

5.2. Detección sobre benchmarks paralelos

Hemos utilizado algunos benchmarks clásicos (ver tabla 5.4), como un ping-pong, kernels de NAS como el IS (integer sort). Y el reciente Intel MPI Benchmark (IMB) que incluye diferentes benchmarks conteniendo diferentes usos de operaciones punto a punto y colectivas; así como algunos benchmarks propios, dedicados a examinar el uso de un gran numero de primitivas dentro de la misma aplicación, como uso de validación amplia de detección de diferentes problemas.

Benchmark Empleado	Problemas encontrados
PingPong (pvm)	5 LS
NAS IS (pvm)	27 LS, 18 BS
IMB-MPI1 p2p (mpi)	3 LS, 1 BB

Tabla 5.4: Resultados de detección sobre benchmarks disponibles

En el caso del benchmark basado en la suite IMB, se han usado solo comunicaciones punto a punto, y algunos de los tests usan únicamente algunos de los procesadores disponibles en el sistema, mientras los otros esperan bloqueados en un Barrier global.

De estos casos podemos observar la importancia relativa de los problemas detectados, en la tabla 5.5, donde encontramos de donde proceden principalmente los tiempos idle ocasionados en la aplicación, seleccionando aquellos problemas más importantes:

Benchmark	Problemas	Idle time (% total time)
PingPong (pvm)	5LS	main 2LS 89 %
NAS IS (pvm)	27LS, 18BS	main 2LS, 1BS 20 %
IMB-MPI1 p2p (mpi)	3LS, 1BB	main 3LS 32 %

Tabla 5.5: Importancia de problemas sobre benchmarks disponibles

En la tabla se indican como *main* aquellos problemas principales que suponen el mayor porcentaje sobre el tiempo total de la aplicación, descartando el resto de problemas por no aportar un porcentaje significativo, teniendo en cuenta que esta selección ya se realiza en la etapa de clasificación de la lista de problemas detectados. Observándose así la fase anterior y posterior de la etapa de clasificación.

5.3. Aplicaciones

En esta parte de la experimentación, las aplicaciones usadas pertenecen a diferentes ámbitos científicos. El objetivo en estos experimentos, es demostrar la

aplicabilidad del modelo de detección, y análisis a aplicaciones reales, y observar que se obtienen recomendaciones útiles para mejorar las prestaciones de las aplicaciones.

Para conducir estos estos experimentos hemos seleccionado una serie de aplicaciones de computo intensivo, en concreto:

1. Una aplicación de simulación de la propagación de incendios forestales Xfire [Jor98, Jor01], desarrollado en nuestro grupo de la Universidad Autónoma de Barcelona. La aplicación Xfire es una aplicación basada en PVM para clusters de de estaciones de trabajo (GNU/Linux o Solaris), donde se simula el avance de la linea de frente de un incendio, para el estudio de su propagación. El estudio parte de la geometría actual del frente del incendio, para evaluar su posible avance considerando los diferentes aspectos climáticos (como viento, temperatura y humedad), vegetación y topografía del terreno.

2. Otra aplicación que se ha usado, es una implementación, desarrollada en PVM, del algoritmo DPMTA (Distributed Parallel Multipole Tree Algorithm) [Ran02]. El propósito de esta implementación es proveer un framework para aplicaciones que necesiten estos algoritmos para computar eficientemente interacciones de tipo N-Body en diferentes tamaños de sistemas de computo, y diferentes configuraciones de numero de partículas.

3. Por ultimo se ha evaluado, una implementación para cálculo de transformadas rápidas de Fourier (FFT) realizada en MPI, denominada FFTW [Fri97]. Básicamente, se trata de una librería C para el cálculo de diferentes variantes directas e inversas de la FFT, para proporcionar soporte a las aplicaciones que la incluyan. Hemos comprobado una aplicación básica que aplicaba las funciones de librería sobre unos conjuntos de datos generados aleatoriamente.

5.3.1. Xfire

En Xfire se simula el avance del frente del incendio, mediante modelos [Jor98], que lo definen como un conjunto de secciones, donde cada sección está compuesta por una serie de puntos (ya sea obtenidos de fotos del incendio, mediciones de campo, mapas, etc.). Cada una de estas secciones ha de ser disgregada del resto para calcular el progreso de cada punto durante el intervalo de tiempo de paso. Cuando el progreso de todos los puntos ha sido calculado, es necesario agregar las posiciones de los diferentes puntos para reconstruir la posición siguiente de la linea del incendio.

Para simular el avance de la propagación, se hacen servir dos modelos: global y local. El modelo global permite la partición del frente en secciones y posteriormente la agregación de las nuevas secciones en la posición siguiente del frente mediante la aplicación de algoritmos numéricos de recomposición de la linea. Con la agregación y el calculo de la nueva posición del frente, este puede expandirse y

ser necesario en ciertas circunstancias añadir puntos a la definición del frente. La definición de las secciones es independiente, pero los puntos finales de cada una de ella son compartidos entre las secciones vecinas. El modelo local calcula el movimiento de cada punto individual. Cuando se evalúa un punto, se usan algoritmos numéricos y son tenidas en cuenta en el modelo numérico las condiciones estáticas y dinámicas (p.e. viento, vegetación, topografía).

La propagación del frente se define en una serie de pasos:

1. Subdivisión del frente en una partición de secciones, con una longitud de puntos determinada.

2. Resolución del problema local en cada sección, dando como resultado un frente virtual de puntos con nuevas posiciones.

3. Agregación del frente virtual, para la definición del nuevo frente en el tiempo del siguiente intervalo.

La simulación de la propagación del incendio requiere de diferentes pasos que suponen cálculos complejos, y por tanto una simulación en detalle, con velocidad del incendio y alta resolución temporal es una computación intensiva en tiempo. La primera implementación de Xfire [Jor98] se realizo con una versión secuencial en PC, pero debido a los limites de prestaciones, y a las necesidades de alto computo, y alto numero de simulaciones a realizar, se decidió implementar una versión paralela en paso de mensajes bajo entorno PVM. Se está utilizando paralelismo de datos, p.e. el calculo de cada movimiento de cada sección (modelo local) se realiza en paralelo. El frente es disgregado en N secciones y cada sección es realizada por una tarea distinta distribuida a lo largo de la maquina paralela o cluster utilizado. Puede realizarse en esta forma por que el modelo considera la evolución de cada sección como independiente. Pero en la etapa de agregación es necesaria la puesta en común de información de limite entre una sección y sus vecinas, proceso que realiza el modelo global de forma centralizada. Xfire es así una aplicación basada en PVM que sigue un paradigma de programación Master Worker. El Master genera las particiones del frente y las distribuye a los workers. Cada worker calcula el modelo local para determinar el avance local. Y el modelo general de la aplicación quedaría como:

Master:

1. Obtener frente inicial.

2. Generar partición del frente en secciones y distribuirlas a los worker.

3. Esperar la respuesta de los worker.

4. Si el tiempo de simulación a acabado acabar, sino componer nuevo frente, añadiendo si es necesario nuevos puntos para aumentar la resolución. Ir a punto 2

Worker:

1. Obtener la sección de frente del Master.

2. Calcular la propagación local de cada punto en la sección (para calcular la nueva posición del punto, el modelo necesita saber sus vecinos derecho e izquierdo)

3. Retornar la nueva sección al master.

Para unas primera pruebas, se tomaron medidas de diferentes de diferentes ejecuciones (mismos datos de entrada) para diferentes configuraciones de nodos de procesamiento. En la siguiente tabla 5.6 se adjuntan datos de escalabilidad de la aplicación, juntamente con datos de las trazas obtenidas, y los tiempos de ejecución obtenidos. En estos tiempos no se incluyen los tiempos correspondientes a arranque de la aplicación, ya que dispone de unos tiempos de arranque constantes relacionados con lecturas de ficheros, y creación estática de estructuras de datos. Obtenemos así solo tiempos de cómputo de los ciclos de simulación. De igual manera en los tiempos con trazador, estos mismos tiempos no están incluidos, y tampoco los correspondientes al arranque del trazador, entre ellos la sincronización de relojes, aproximadamente unos (constantes) 30 segundos en el caso del trazador TapePVM que nos ocupa), y tampoco la fusión final de las trazas individuales en una de sola (aproximadamente entre 10 a 30 segundos más dependiendo del tamaño).

Nodos esclavos	tiempo ejecución	tiempo con trazador	Tamaño traza (MB)
1	4min05sec	4min16sec	0,46
2	2min23sec	2min30sec	0,92
4	1min46sec	1min48sec	1,80
8	1min40sec	1min39sec	3,57
16	1min30sec	1min33sec	7,05

Tabla 5.6: Pruebas de ejecución de Xfire

Las diferencias de tiempo observadas en 5.6 para una misma configuración, solamente son atribuibles a la existencia del trazador y su perturbación de la ejecución, ya que se han eliminados las fases (más o menos constantes) de sincronización inicial de relojes, y de compilación de traza final. Se observa que esta perturbación es mínima, siendo solamente un porcentaje aproximado de 5 % respecto la ejecución sin trazador. Así mismo se observa en algún caso el efecto contrario, una disminución de tiempo, debido a que precisamente, un cierto retardo de tiempo puede provocar que una comunicación se sincronice mejor que si no existiese (casos por ejemplo de emisor y/o receptor en un Late Sender o un Blocked Sender por ejemplo), este mismo efecto puede provocar una sincronización mejor a posteriori del resto de primitivas de paso de mensajes tanto punto a punto como colectivas.

En la siguiente tabla 5.7 se observan las ineficiencias encontradas en las diversas pruebas, junto con las detectadas como principales, y el tiempo de la herramienta en el proceso (solo se incluye detección y clasificación de ineficiencias).

Esclavos	Ineficiencias encontradas	Principales	Tiempo análisis
1	6 LS	1 LS	445 ms
2	2 BS, 7 LS	3 LS	555 ms
4	4 BS, 10 LS	3 LS	797 ms
8	9 BS, 20 LS	2 LS	1313 ms
16	74 BS, 45 LS	10 LS	2036 ms

Tabla 5.7: Ineficiencias detectadas en Xfire

En las aplicaciones reales como xfire, podemos encontrar habitualmente un mayor numero de problemas detectados (no siempre, dependerá de la complejidad de la aplicación y de su traslación al paradigma de paso de mensajes). Ya que usualmente delante de una aplicación de tamaño medio-grande, el desarrollador no puede comprender completamente la complejidad de las interrelaciones entre tareas, procesadores, red y patrones de comunicación, y esto causa la aparición de grupos mayores de ineficiencias, algunas causadas por esta falta de comprensión, otras por la no correlación directa de modelos de aplicación y algoritmo con el sistema físico disponible.

La herramienta, en el caso del análisis de Xfire (ver tabla anterior 5.7), sugiere un incremento importante de los problemas, en los casos de 8 y 16 procesadores. En el análisis detallado se comprueba que los casos de LS son causados básicamente en la espera del master sobre los datos de los esclavos, mientras están computando. Aunque hay un aumento de blocked Senders importante, teniendo el master por medio. En este caso el análisis de estructuras de tipo master-worked detecto que el master estaba creando la mayor parte de los problemas, y en concreto se detecto que el aumento importante de LS y BS eran causados tanto en el caso de 8 y 16 procesadores, era causado por la existencia de una tarea esclava en el mismo nodo de computo que el master. Se sugirió evitar esta disposición indicando la zona del código donde se realizaba la creación de esclavos.

En este caso, Xfire dejaba al propio PVM la asignación de tareas a nodos. En un caso los peores resultados tenían sentido, debido (en el caso de 16) al uso de tan solo las 16 máquinas inicialmente disponibles en el cluster. Pero en el caso de 8, era el propio asignador de PVM que fallaba, o detectaba fallos en los nodos, y no utilizaba el nodo previsto para asignar inicialmente, colocando el esclavo en el mismo nodo inicial que el master.

La solución tomada fue incorporar una lista fija de nodos, y realizar la asignación manualmente (en el caso de 16, se empleo una máquina más disponible, 17 en total). Se obtuvieron las siguientes mejoras para los casos problemáticos:

Que supusieron una mejora respectivamente del 11 y 10 % respecto de la situación inicial causada por las problematicas de master y esclavo en el mismo nodo.

Esclavos	tiempo ejecución
8	1m29sec
16	1m21sec

Tabla 5.8: Asignación manual de nodos en Xfire

5.3.2. DPMTA

Esta aplicación [Ran02] es una versión de calculo distribuido de los algoritmos de multipolos, en particular del FMA (Fast Multipole Algorithm), el cual es un algoritmo numérico para el problema de N-Body.

Este problema clásico, envuelve la computación del efecto en red de las interacciones de cada par de partículas dentro del conjunto de N partículas iniciales. Estos algoritmos son de aplicación en muchos campos, por ejemplo en dinámica molecular, donde las partículas son átomos y las fuerzas electroestaticas. O otro campo, como por ejemplo las simulaciones astrofísicas, donde las partículas son cuerpos estelares y las fuerzas gravitacionales.

En cualquier de estos casos, las computaciones necesarias crecen con el cuadro del numero de partículas, para las implementaciones comunes. Para disminuir este efecto las implementaciones FMA usan lo que se denomina expansión de multipolos, intentando representar los efectos de las partículas distantes como una entidad simple. Utilizando este mecanismo, y combinando estos efectos multipolo, pueden obtenerse computaciones que pueden reducirse a modelos casi lineales con relación al numero de partículas.

Finalmente en el caso de los PMTA se usa este concepto de multipolo, con la representación de la descomposición espacial mediante arboles octales. El cual es es caso de esta implementación distribuida (DPTMA).

Esta implementación esta pensada para incluirla en cualquier programa (por ejemplo de tipo simulador) que necesite estos cálculos, de manera que este le pase un conjunto de datos de partículas, y se le retorne los cómputos de todas las interacciones de los pares, conjuntamente con los datos de las fuerzas y potenciales evaluados. En esta caso de experimentación se ha evaluado un caso de prueba (dpmta_test) que incluye de ejemplo la implementación mencionada.

Respecto al modelo de alto nivel de la implementación destacamos en los siguientes puntos, la estructura de la computación paralela [Ran02]:

Paso 0 Se procede a la inicialización de los procesos esclavos. El proceso master arranca todos los esclavos requeridos. Los esclavos entonces computan sus listas de interacciones con los vecinos, y a su vez las listas inversas hacia ellos. Cada esclavo usa memoria estática para una serie de arrays constantes para los cálculos de los multipolos, y pide memoria dinámica para las estructuras de datos particulares de las partículas y celdas espaciales.

Paso 1 Descomposición de partículas. El master envía los datos de posiciones de partículas a cada proceso esclavo. Cada esclavo recibe datos solo de las

partículas en las celdas de las que es propietario.

Paso 2 Redistribución de partículas. Cada esclavo envía datos de partículas a los otros procesos esclavos que requieren los datos para sus propios cálculos. Los contenidos de los mensajes son determinados por las listas inversas de interacción enviadas a cada esclavo.

Paso 3 Calculo de las expansiones de los multipolos. Para cada celda en el nivel más bajo del árbol, el proceso esclavo computa los multipolos en el centro del espacio debido a todas las partículas dentro del espacio. No se necesita ni comunicación ni sincronización en este punto, debido a que cada proceso puede realizar los cálculos con los datos ya asignados a cada esclavo.

Paso 4 Paso hacia arriba. Desde el nivel más bajo del árbol, el multipolo calculado en el centro de la celda asignada al esclavo se desplaza al de la celda padre, y los resultados son acumulados para la celda de nivel superior. Si la celda padre no esta asignada al esclavo actual, este envía el calculo al proceso esclavo que tiene asignada la celda padre.

Paso 5 Recepción de partículas. Cada esclavo recibe y guarda la información de partículas recibidas desde el paso 2.

Paso 6 Distribución de multipolos. Cada esclavo envía los datos de los pasos 2 y 3 a los otros esclavos que requieren los datos. Según las listas inversas.

Paso 7 Computación directa. Cada partícula debe interactuar directamente con las partículas en las celdas (definidas como vecinas) especificadas en sus listas de interacción. Por cada celda que el esclavo tiene, se computan las interacciones entre pares de partículas con las otras celdas que el esclavo posee, actualizando los vectores de la fuerza y el potencial para las partículas de ambas celdas. Para las que no sean del mismo esclavo solo se actualizan los valores calculados localmente.

Paso 8 Recepción de multipolos. Cada esclavo recibe y guarda la expansión de multipolos enviada en el paso 6.

Paso 9 Paso hacia abajo. Cada celda obtiene la información local de multipolos desde su celda padre, y calcula sus expansiones basadas en las del padre y las de cada celda en su lista de interacciones. Si la celda hija pertenece a un esclavo diferente, la información se pasa al esclavo pertinente.

Paso 10 Evaluación de la expansión local. Cada proceso esclavo evalúa su expansión local para cada partícula del nivel más bajo del árbol. Estas fuerzas resultantes son añadidas a las calculadas por el método directo para dar finalmente una fuerza final a cada partícula resultante de las todas las otras partículas de la región simulada.

Paso 11 Recolección de resultados. Cada esclavo envía la información de fuerzas para sus partículas de vuelta al master que las recolecta, concluyendo el ciclo.

La clave en esta implementación para la obtención de prestaciones del algoritmo es la simultaneidad de paso de mensajes con computación. De esta manera, como ya vimos, el efecto de la latencia en el paso de mensajes, puede ser ocultada.

En este algoritmo en particular, el uso de listas inversas permite a los esclavos, conocer que datos serán necesarios para otros procesadores, para así poderlos enviar cuando estén disponibles, sin tener que responder a una petición previa de estos.

Para nuestros experimentos con la implementación DPMTA hemos ejecutado una serie de pruebas iniciales que nos han permitido primero examinar la escalabilidad de la implementación delante de diferentes configuraciones de sistema y conjuntos de datos, en este caso numero de partículas empleadas. En la tabla siguiente se resumen los experimentos iniciales 5.9 teniendo en cuenta diferentes configuraciones de máquinas, y algunos conjuntos de números de partículas representativos.

Nodos	N	Tiempo ejecución	Ejecución Trazador	Traza (MB)
2	1024	0min17sec	1min36sec	0.04
4	1024	0min13sec	1min28sec	0.16
8	1024	0min13sec	1min29sec	0.65
16	1024	0min16sec	1min36sec	2.60
2	10240	2min10sec	3min26sec	0.04
4	10240	1min24sec	2min36sec	0.16
8	10240	1min05sec	2min12sec	0.65
16	10240	0min56sec	2min07sec	2.60
2	30240	16min8sec	17min32sec	0.04
4	30240	9min31sec	10min46sec	0.16
8	30240	6min34sec	7min48sec	0.65
16	30240	5min40sec	6min09sec	2.60

Tabla 5.9: Pruebas escalabilidad de la implementación DPMTA

En el caso de la presente tabla, el trazador usado (TapePVM) tiene a lo largo de la experimentación, unos tiempos constantes, un periodo constante de 30 segundos de sincronización de relojes (configurable), más un tiempo de recolección de eventos entre 20 a 30 segundos según tamaño de trazas, lo que comporta en el caso de nuestras pruebas, aproximadamente 1 minuto extra por prueba, el tiempo extra es el causado por las perturbaciones durante la ejecución. Estas perturbaciones volverían a estar según los datos entre un 4 % a 6 %.

Como ya sucedió con Xfire, en algunos casos, las perturbaciones introducidas pueden llevar a dar mejores resultados por mejorar las sincronizaciones y/o comunicaciones puntuales.

Si se aplica el análisis de la herramienta a uno de los conjuntos de datos, por ejemplo al tamaño 30240, observamos las siguientes ineficiencias encontradas (tabla 5.10):

Numero nodos	ineficiencias	Principales	Tiempo análisis
2	7 LS	2 LS	285 ms
4	18 BS, 30 LS	5 LS	388 ms
8	89 BS, 103 LS	4 LS	577 ms
16	84 BS, 154 LS	10 LS	1285 ms

Tabla 5.10: Ineficiencias detectadas en experimentos DPMTA

En la tabla encontramos un numero creciente de LS a medida que incrementamos el numero de procesadores, y que podemos interpretar debido a las relaciones de vecindad necesarias para el cálculo, en las diferentes iteraciones, y BS relacionados a priori con los mecanismos de comunicación por niveles que tenemos en la aplicación. Los problemas principales se han estabilizado a excepción del ultimo caso, donde han aumentado de forma significativa.

En el análisis posterior la herramienta a detectado diversas zonas problemáticas en el código relacionadas con los LS mencionados. Además en el último caso (16) se daba también una situación parecida a Xfire, teníamos 16 esclavos, donde uno compartía nodo con el proceso master. Respecto al análisis de código se encontraron varias relaciones envió y recepción correspondientes a los LS, pero casi todas ellas tenían dependencias fuertes de datos, que impedían el avance de datos, se detecto únicamente la posibilidad de retrasar algunas recepciones asociadas a los LS, avanzando código no dependiente a los primitivas de recepción (pvm_send en este caso).

En las ejecuciones posteriores (tabla 5.11), sobre el tamaño mayor de datos, se obtuvieron los siguientes resultados:

Numero nodos	Tiempo nueva ejecución
2	16min9sec
4	9min33sec
8	6min30sec
16	5min20sec

Tabla 5.11: Resultados de las nuevas ejecuciones de DPMTA

En este caso debido a las fuertes dependencias de los datos enviados y recibidos para el computo posterior, y a la estratificación de la aplicación en unas capas muy definidas, impidió mayores optimizaciones. Los resultados obtenidos sugieren unas mejoras casi despreciables, a excepción del problema del mapping del caso de 16 esclavos.

5.3.3. FFTW

La aplicación usada realiza una serie de cálculos FFT, usando una versión paralela de la librería FFTW. Esta librería implementa transformadas multi-dimensionales de datos reales y complejos. De forma que los datos de la transformada son repartidos entre las máquinas disponibles, recibiendo cada proceso una porción del array.

La implementación de FFTW en MPI, incluye diversos tests, entre los cuales se ha analizado un test que realizada varias transformadas sobre matrices cuadradas.

Un proceso, normalmente el de rank MPI 0, sera el encargado de pedir espacio para el almacenamiento de los datos, y inicializar los datos. Que serán posteriormente distribuidos entre los procesos envueltos en la realización de la transformada.

En concreto el array es dividido por filas de los datos, y cada proceso obtiene un subconjunto de las filas de los datos. En particular es interesante, para las prestaciones que las dimensiones del array sean divisibles por el numero de procesadores empleados, para que pueda darse un tratamiento homogéneo de datos entre procesadores. Además, tenemos que tener en cuenta que se sigue el mismo mecanismo básico para la división, con lo cual la variación del numero de procesadores afecta directamente a la relación de comunicación y computo (para un tamaño de datos prefijado).

En la tabla siguiente 5.12 pueden verse diferentes experimentos realizados con la aplicación usada para probar las rutinas FFTW (sobre un tamaño de array de 10000x10000):

Numero nodos	tiempo ejecución	tiempo trazador
4	4min25sec	4min39sec
8	3min3sec	3min16sec
16	2min30sec	2min43sec

Tabla 5.12: Pruebas FFTW para matriz grande

En general, en esta aplicación, como comentamos se nota la variación de prestaciones dependiendo del ratio de cómputo respecto comunicaciones, ya que el patrón de intercambio es siempre el mismo, solamente cambia el tamaño de los mensajes intercambiados.

Al aplicar el Trazador MPITracer (como comentamos en el capítulo), observamos un incremento de tiempo entre un 5 % a 8 % (similar a los generales del trazador [Iva04]), siempre sin evaluar los tiempos de sincronización de relojes inicial y recolección final de traza, que no afectan directamente al desarrollo de la ejecución.

Respecto a la traza obtenida, esta es de 500 KB aproximadamente, en particular se mantiene reducida, por el efecto de llamadas a paso de mensajes comunes a todos los casos, solamente hay variaciones del tamaño de los datos enviados en cada llamada, y siempre que se mantenga divisibilidad de los tamaños de datos entre el numero de los procesadores, supone un tratamiento homogéneo de la aplicación.

En el análisis la herramienta ha detectado 3 casos de Blocked Barrier, 4 Colectiva 1 a N, y 10 Colectivas NxN. Siendo significativas 2 ineficiencias NxN y 3 Colectivas 1 a N. En relación con el código la herramienta detecto que las colectivas afectaban a todos los procesos participantes (en este caso se utiliza comunicadores globales), y se creaban tiempos de espera altos en las NxN (estas no podían moverse debido a dependencias de datos), este caso el problema anterior importante en el tiempo era el de las ineficiencias colectivas 1 a N, este caso se relaciono directamente en el código por una serie de 3 Broadcasts consecutivos que se sugirio optimizar.

Al inspeccionar el código, se pudo comprobar que así era, de hecho en el código aparecían 3 llamadas consecutivas de envió de datos mediante broadcast a los mismos participantes. Decidimos agrupar los datos en un único envió mediante broadcast. En este mismo caso se pudo comprobar manualmente, que existía 1 Broadcast extra previo (la 4 ineficiencia colectiva, que había evaluado previamente la herramienta), que no tenia dependencia de datos, y podía también agruparse con los 3 anteriores. De esta manera se fusiono los cuatro Broadcasts originales a una sola llamada.

Los resultados posteriores fueron:

Numero nodos	tiempo ejecución
4	4min18sec
8	2min45sec
16	2min13sec

Tabla 5.13: Pruebas FFTW con optimización primitivas iniciales

Con lo que obtenemos aproximadamente un 9 % de mejora sobre la ejecución inicial. En este ejemplo hay que tener en cuenta la rigidez del modelo basado unicamente en primitivas colectivas con fuertes dependencias de datos entre los envios y las fases de cálculos.

5.4. Conclusiones

Los diferentes ámbitos de experimentación, han permitido validar las diversas fases de la herramienta, así como comprobar su utilidad de cara afrontar el análisis automático de aplicaciones científicas mayores, con sugerencias emitidas de cara al usuario final, y que han permitido mejorar, su uso y rendimiento en el sistema físico utilizado.

En particular puede comprobarse que incluso los problemas más simples de prestaciones pueden causar tiempos importantes de inactividad, causando las consecuentes perdidas de prestaciones.

Cabe destacar la validación usando los benchmarks construidos mediante el conjunto de herramientas proporcionado por el APART Testsuite, que a permitido una validación de la especificación de los problemas simples, y por consecuencia

de la correcta detección de estos en base a su especificación. Así como la detección de problemas compuestos por combinaciones de los problemas expuestos.

Esta aproximación de benchmarking es muy oportuna para la validación de herramientas automáticas, y permitirá la construcción de benchmarks más amplios que permitan testear la cobertura, de una determinada herramienta, enfrente de un conjunto de problemas, o conjunto de benchmarks con diferentes problemas. Esto nos permite poder comparar las herramientas, al menos en su fase de detección. Para las subsiguientes fases todo depende de las aproximaciones tomadas por las diferentes herramientas, muchas de ellas no equivalentes, o con diferentes modelos de prestaciones, que no tienen porque llevar a un mismo fin, dependiendo de las métricas de prestaciones utilizadas, ya sea mejorar el tiempo de ejecución total, la escalabilidad, el balanceo de carga de los nodos del sistema, la eficiencia de los recursos, o cualquier otra medida o función del conjunto de estas que se considere óptimo en el modelo de prestaciones escogido.

En nuestro caso, se ha demostrado que se obtienen resultados validos, para afrontar la reducción del tiempo total de ejecución, tanto en los casos de benchmarks orientados a problemas concretos, kernels de aplicación, o aplicaciones completas.

Capítulo 6

Conclusiones y lineas abiertas

Este capítulo presenta las principales conclusiones y experiencias derivadas del estudio realizado en esta tesis y las propuestas de futuras líneas de investigación derivadas del trabajo.

6.1. Conclusiones

En este trabajo hemos propuesto y desarrollado una solución automática para las problemáticas que comporta el análisis de prestaciones de aplicaciones paralelas y/o distribuidas basadas en modelos de paso de mensajes.

Nuestra investigación en este trabajo, se ha dirigido la concepción de una arquitectura abierta para el diseño y desarrollo de herramientas de análisis automático de prestaciones. Las herramientas diseñadas siguiendo esta especificación toman como entrada el conocimiento disponible sobre problemas de prestaciones en entornos de paso de mensajes, y permiten usarlo para la mejora del proceso de sintonización de las aplicaciones paralelas/distribuidas por medio de sugerencias dirigidas a un usuario no experto en todos los campos necesarios, dada la complejidad inherente al desarrollo, implementación y mantenimiento de las aplicaciones paralelas.

Este objetivo se ha plasmado en el diseño de una arquitectura que cumple una serie de premisas: Aportación (abierta) de conocimiento sobre problemas de prestaciones en paradigmas de paso de mensajes, abstracción del modelo de paso de mensajes para incorporar diferentes entornos existentes, y generación de sugerencias directamente utilizables por el desarrollador final sobre como modificar su aplicación para obtener mejores índices de rendimiento sobre los sistemas finales.

El requisito de la arquitectura, sobre aportación de conocimiento se ha visto cumplido gracias al uso de un lenguaje de especificación de los problemas de prestaciones, en forma de patrones estructurales de eventos.

La abstracción de modelos de paso de mensajes nos ha proporcionado independencia de la arquitectura respecto a las diferentes implantaciones de paso de

mensajes. Pudiendo trabajar sobre implementaciones de PVM o MPI indistintamente, y con las posibles variaciones de cada fabricante. La arquitectura se ha basado solamente en modelos de las primitivas habituales de paso de mensajes, y de los tiempos involucrados en éstas.

Las sugerencias emitidas hacia el desarrollador, nos permiten hacer comprensibles las perdidas de prestaciones de la aplicación, al nivel que el desarrollador trabaja, mostrándole en su código directamente, aquellas zonas, regiones que se han visto involucradas, y proporcionándole sugerencias de los cambios a realizar para aportar la posibilidad de eliminar o evitar los problemas que han surgido durante la ejecución de la aplicación.

Todas estos requerimientos de la arquitectura se han plasmado en el diseño e implementación de una herramienta de análisis automático de prestaciones, denominada KappaPI 2 (Knowledge-based Automatic Parallel Program Analyser for Performance Improvement). Como prueba de concepto, esta herramienta demuestra la validez de la arquitectura propuesta, y la viabilidad de incorporar conocimiento sobre nuevos problemas de prestaciones, así como del proceso de análisis automático y generación de recomendaciones encaminadas a corregir los problemas de prestaciones significativos detectados.

KappaPI 2 ha sido verificada con un amplio rango de aplicaciones, incluyendo aplicaciones sintéticas (basadas en el APART testsuite), benchmarks paralelos, y aplicaciones reales. Las medidas cuantitativas obtenidas de las diferentes pruebas experimentales, sobre la arquitectura de la herramienta, corroboran que la detección de problemas de prestaciones en entornos de paso de mensajes, pueden basarse en patrones estructurales de eventos. En particular el uso de una herramienta, como KappaPI 2, abierta a la incorporación de conocimiento, ofrece un entorno de trabajo potente y flexible a su vez, para incorporar el conocimiento necesario para guiar el proceso automático de detección de los problemas. La emisión de sugerencias directas orientadas al desarrollador, es una aproximación que proporciona una ayuda muy significativa en el proceso de sintonización de la aplicación, mediante información directamente interpretable para el desarrollador.

El desarrollo del trabajo, a pasado por diferentes fases, desde el interés inicial en las aplicaciones paralelas, al objetivo final de conseguir altas prestaciones a través del análisis automático de prestaciones. Respecto a nuestro trabajo previo al de esta tesis cabe destacar el paso por el diseño e implementación de aplicaciones paralelas en el campo científico, que nos sirvió de base para estudiar las diferentes problemáticas a que el desarrollador tiene que enfrentarse, tanto a las optimizaciones iniciales, como al análisis de prestaciones manual, tanto en los roles de desarrollador conocedor de la aplicación, como de usuario de herramientas de visualización.

En este primer estudio desarrollamos la aplicación Xfire (ya comentada en los casos de estudio), que nos permitió atacar mediante paralelismo uno de los campos más preocupantes actualmente como problema medioambiental (los incendios forestales). En este trabajo observamos las limitaciones que tenían las herramientas visuales para poder examinar donde estaban realmente los problemas, y diri-

gir correctamente su resolución. En este campo desarrollamos varios trabajos tanto desde el punto de vista medioambiental (implementación de modelos físicos) [Jor98] como desde el punto de vista de paralelismo y optimización manual de prestaciones [Jor99, Jor01].

[Jor98] Josep Jorba, Tomas Margalef, Emilio Luque, J. Andre, and D. Viegas. Application of parallel processing to the simulation of forest fire propagation. In Proceedings of the III International Conference on Forest Fire Research, volume II, pages 891 a 900, Coimbra, Portugal, 1998.

[Jor99] Josep Jorba, Tomas Margalef, and Emilio Luque. Simulación paralela de propagación de incendios forestales. In Proceedings de las X Jornadas de Paralelismo, 1999.

[Jor01] Josep Jorba, Tomas Margalef, and Emilio Luque. Simulation of forest fire propagation on parallel and distributed pvm platforms. Lecture Notes in Computer Science, 2131:386, January 2001.

Destacar también que esta aplicación ha servido de base a numerosos estudios de prestaciones en las diversas herramientas de prestaciones realizadas en nuestro grupo, tanto en el campo del análisis estático en la primera versión de KappaPI o en la actual herramienta de segunda generación KappaPI 2, como en el análisis dinámico de prestaciones en un entorno como MATE [Mar04].

[Mar04] Tomas Margalef, Josep Jorba, Oleg Morajko, Anna Morajko, and Emilio Luque. Performance Analysis and Grid Computing, chapter Different approaches to automatic performance analysis of distributed applications, pages 3 a 19. Kluwer Academic Publishers, Norwell, MA, USA, 2004.

Respecto al análisis de prestaciones basado en herramientas automáticas, destacar la realización de estudios de diferentes soluciones [Mar04] y trabajos complementarios respecto la herramienta KappaPI 2, enfocados al análisis dinámico de prestaciones [Mor03, Mor03a], que nos aportaron la visión complementaria del otro segmento de herramientas, así como las limitaciones y el rango de aplicación de cada tipo de herramientas.

[Mor03] Anna Morajko, Oleg Morajko, Josep Jorba, Tomas Margalef, and Emilio Luque. Automatic performance analysis and dynamic tuning of distributed applications. Parallel Processing Letters, 13(2):169 a 187, 2003.

[Mor03a] Anna Morajko, Oleg Morajko, Josep Jorba, Tomas Margalef, and Emilio Luque. Dynamic performance tuning of distributed programming libraries. Lecture Notes in Computer Science, 2660:191 a 200, January 2003.

En cuanto a la herramienta actual, su propuesta de diseño, y resultados de las primeras pruebas, fueron mostradas en una reunión del proyecto APART [Jor02] (Junio 2002, Barcelona):

[Jor02] Josep Jorba. Static performance analysis: Kappapi 2 state of art (apart project 2002 meeting notes). `http://www.fz-juelich.de/apart-2/meet5/kpi2.ppt` , June 2002.

A partir de la propuesta de diseño inicial, se realizaron diferentes experiencias con la especificación de los problemas, y la redefinición de las fases de la herramienta que se publicaron en [Mar04], juntamente con los estudios complementarios realizados en el campo del análisis automático de tipo dinámico de prestaciones, para la herramienta MATE.

[Mar04] Tomas Margalef, Josep Jorba, Oleg Morajko, Anna Morajko, and Emilio Luque. Performance Analysis and Grid Computing, chapter Different approaches to automatic performance analysis of distributed applications, pages 3 a 19. Kluwer Academic Publishers, Norwell, MA, USA, 2004.

Posteriormente se completaron las diferentes fases de definición de las etapas finales de la herramienta orientadas al análisis final del problema de los casos, y del análisis del código fuente de la herramienta que fueron publicados en los trabajos [Jor05] [Jor05a]:

[Jor05] Josep Jorba, Tomas Margalef, and Emilio Luque. Automatic performnace analysis of message passing applications using the kappapi 2 tool. Lecture Notes in Computer Science, 3666:191 a 200, September 2005.

[Jor05a] Josep Jorba, Tomas Margalef, and Emilio Luque. Performance analysis of parallel applications with kappapi 2. In Procedings of PARCO 2005 Conference, 2005.

Posteriormente a estos trabajos, se han refinado los elementos del análisis de código, y ampliado los experimentos realizados, así como algunas de las especificaciones de los problemas, y su proceso de análisis, según se ha mostrado en esta tesis.

6.2. Líneas abiertas

De la experiencia obtenida durante la realización de este trabajo han surgido nuevos retos y líneas de futuros trabajos, que a continuación enumeramos:

- Ampliaciones relacionadas con el grado de cobertura de las APIs de paso de mensajes. En la implementación actual están evaluados eventos de primitivas habituales, como p2p y colectivas generales. Podrían incluirse primitivas más especificas, y en particular algunas relacionadas con características de los últimos estandars (como MPI2).

- Mejora de las operaciones de unión y solapamiento de problemas. En muchos casos los problemas encontrados, sugieren una aparición simultanea o

complementaria de unos a otros, en algunos casos es difícil evaluar la influencia de la fusión o unión de estos problemas en las prestaciones globales de la aplicación. En algunos casos podría ser necesaria una simulación de las diferentes alternativas de su unión o solapamiento, en particular cuando los problemas implicados fueran varios, y pudieran existir dependencias entre los mismos. Esto podría tratarse con aproximaciones basadas en camino crítico dentro de la aplicación, y en particular si se proponen alteraciones de patrones de comunicación, como alterar tareas participantes, o cambiar modelos p2p a colectivas o viceversa.

- En el análisis de código fuente, hemos incorporado en la arquitectura una aproximación basada en una API simple con una funcionalidad limitada como compromiso entre posibilidades ofrecidas respecto complejidad, y resultados obtenidos. Una mejora clara a introducir, es la incorporación de entornos más complejos de análisis de código. Algunas de las opciones disponibles, para utilizar como sistema base, podrían ser los entornos PUMA, SUIF i Sage++.

- Introducción de entorno multiexperimento para la herramienta, para que esta pudiese disponer del histórico de ejecuciones y la variabilidad de los datos de prestaciones dependiendo de las ejecuciones. Para determinados tipos de aplicaciones fuertemente dependientes de los datos (de su tamaño, distribución, etc...) así como de aquellas que incluyan varios algoritmos, y el flujo de la aplicación pueda depender de ellos; en estos casos sería muy interesante poder disponer de este tipo de análisis. En principio, la inclusión parece viable, ya que se podría incorporar en el momento de clasificación de los problemas consultando, por ejemplo, como otro factor, la historia de ejecuciones. En este caso entraríamos dentro de un aprendizaje de la herramienta frente a un histórico de ejecuciones. Trabajos como los de Karavanic [Kar97] en el proyecto Paradyn, o los realizados en Aksum [Fah02], o bien entornos multiexperimento para predicción de prestaciones como los de Prophesy [Wu00, Wu01, Wu01a], o Prophet [Pll05] podrían servir de guia en este aspecto.

- Relacionado con el anterior, podríamos completar el bucle de análisis automático de la herramienta, con una fase extra de la herramienta, que permitiese hacer cambios realmente automáticos sobre el código fuente, según las sugerencias realizadas por la herramienta (en la medida de lo posible). De este modo, la herramienta volvería a realizar la ejecución sobre los cambios propuestos, y examinaría así las prestaciones finales obtenidas, y dedujera si los cambios realizados han sido útiles o no, y lo pusiese en relación al histórico. En este sentido seria útil un esquema de gestión de versiones del código de la aplicación en árbol, y operaciones de hacer/rehacer cambios sobre el código.

- Otro esquema, sería el de validar por estimación temporal los posibles cambios a introducir, en la fase de análisis. Tenemos el análisis de los tiempos de inactividad causados, podríamos estimar que significaría alterar el problema según una sugerencia hecha, y estimar el porcentaje de mejora que se podría obtener. Los trabajos de modelización, simulación y evaluación de comunicaciones en paso de mensajes mediante herramientas con estrategias preejecución, como las mostradas en la sección 3.1 podrían dar una base para iniciar estos aspectos.

- La detección de estructuras de programación, es un punto interesante a ampliar debido al uso mas o menos intensivo que de ellas se hace en las aplicaciones, al estar basadas en paradigmas de programación, permiten captar el modelo de la computación que se realiza, y llegar a un nivel más alto de comprensión de los problemas de prestaciones, que el paralelismo explícito de primitivas. En particular, es interesante la posible detección de múltiples paradigmas de programación en la aplicación. Por ejemplo, el uso de fases de Master Worker, seguidas de fases de pipeline, u otras. Definiendo así incluso un nivel más, donde se puede capturar el modelo de la aplicación a lo largo del tiempo de su ejecución, e identificar las fases de paradigmas por las que pasa, así como el que provoque menor rendimiento.

- Pueden incorporarse más detecciones de estructuras de paradigmas, mediante la inclusión de SPMD, árbol (por ejemplo en divide-conquer), mallas, anillos, que nos pueden aportar la posibilidad de realizar un análisis extra de mayor nivel entre las interacciones entre tareas y las relaciones entre cómputo y comunicaciones.

Bibliografía

[APA99] ESPRIT Working Group APART. Automatic performance analysis: Resources and tools. `http://www.fz-juelich.de/apart/`, 1999.

[Afr99] S. Afroz, Hee Yong Youn, and Dongman Lee. Performance of message logging protocols for nows with mpi. In *Dependable Computing, 1999. Proceedings. 1999 Pacific Rim International Symposium on*, pages 252–259, 1999.

[Akl89] S. G. Akl. *The Design and Analysis of Parallel Algorithms*. Prentice-Hall, Englewood Cliffs, NJ, 1989.

[Alb99] C. Albrecht, C. D. Pham, and Universite Claude Bernard Lyon I. Optimizing message aggregation for parallel simulation on high performance clusters, aug 1999.

[Alm03] F. Almeida, D. Gonzalez, L.M. Moreno, C. Rodriguez, and J. Toledo. On the prediction of master-slave algorithms over heterogeneous clusters. In *Parallel, Distributed and Network-Based Processing, 2003. Proceedings. Eleventh Euromicro Conference on*, pages 433–437, 2003.

[Alm94] George S. Almasi and Allan Gottlieb. *Highly Parallel Computing, 2nd ed.* Benjamin/Cummings division of Addison Wesley Inc., Redwood City, CA, 1st edition, 1994.

[Als96] Ehab S. Alshaer, Hussein Abdel wahab, and Kurt Maly. High performance monitoring architecture for large scale distributed systems using event filtering, September 07 1996.

[Bag01] Rajive Bagrodia, Ewa Deelman, and Thomas Phan. Parallel Simulation of Large-Scale Parallel Applications. *International Journal of High Performance Computing Applications*, 15(1):3–12, 2001.

[Bag98] Rajive L. Bagrodia and Sundeep Prakash. Mpi-sim: Using parallel simulation to evaluate mpi programs, October 16 1998.

[Bal96] Thomas Ball and James R. Larus. Efficient path profiling. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 46–57, Paris, December 2–4, 1996. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[Ban04] Cyril Banino, Olivier Beaumont, Larry Carter, Jeanne Ferrante, Arnaud Legrand, and Yves Robert. Scheduling strategies for master-slave tasking on heterogeneous processor platforms. *IEEE Transactions on Parallel and Distributed Systems*, 15(4):319–330, April 2004.

[Ban99] Mohammad Banikazemi, Rama K Govindaraju, Robert Blackmore, and Dhabaleswar K Panda. Implementing efficient MPI on LAPI for IBM RS/6000 SP systems: Experiences and performance evcaluation. In *Proceedings of the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing, IPPS/SPDP'99 (San Juan, Puerto Rico, April 12-16, 1999)*, pages 183–190, Los Alamitos-Washington-Brussels-Tokyo, 1999. IEEE Computer Society, ACM SIGARCH, IEEE Computer Society Press.

[Bat86] Peter Charles Bates. *Debugging Programs in a Distributed System Environment*. PhD thesis, University of Massachusetts, February 1986. (Also published as Technical Report 86-05, Department of Computer and Information Science, University of Massachusetts, Amherst, MA 01003, February 1986.).

[Bat95] Peter C. Bates. Debugging heterogeneous distributed systems using event-based models of behavior. *ACM Trans. Comput. Syst.*, 13(1):1–31, 1995.

[Ben90] Moti Ben-Ari. *Principles of Concurrent Programming*. Prentice-Hall International, Inc., Englewood Cliffs, 1990.

[Ber01] M. Bernaschi and G. Richelli. MPI collective communication operations on large shared memory systems. In Bob Werner, editor, *Proceedings of the Ninth Euromicro Workshop on Parallel and Distributed Processing*, pages 159–164, Los Alamitos, California, February 7–9 2001. IEEE Computer Society.

[Ber04] Andre R. Bernat and Barton P. Miller. Incremental call-path profiling. Technical report, University of Wisconsin, Feb 2004.

[Bha05] N. Bhatia, Fengguang Song, F. Wolf, J. Dongarra, B. Mohr, and S. Moore. Automatic experimental analysis of communication patterns in virtual topologies. In *Parallel Processing, 2005. ICPP 2005. International Conference on*, pages 465–472, 2005.

[Blo00] D. M. Blough and P. Liu. FIMD-MPI: A tool for injecting faults into MPI applications. In *14th International Parallel and Distributed Processing Symposium (SPDP'2000)*, pages 241–250, Washington - Brussels - Tokyo, May 2000. IEEE.

[Blo95] Robert J. Block, Sekhar Sarukkai, and Pankaj Mehra. Automated performance prediction of message-passing parallel programs. In *Proceedings of Supercomputing'95*, San Diego, CA, December 1995. ACM/IEEE.

[Boe03] Cristina Boeres and Vinod E. F. Rebello. Towards optimal static task scheduling for realistic machine models: Theory and practice. *International Journal of High Performance Computing Applications*, 17(2):173–189, 2003.

[Bro00] J. C. Browne, E. Berger, and A. Dube. Compositional Development of Performance Models in Poems. *International Journal of High Performance Computing Applications*, 14(4):283–291, 2000.

[Buc00] Bryan Buck and Jeffrey K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[Buy00] Rajkumar Buyya. PARMON: A portable and scalable monitoring system for clusters, September 21 2000.

[Buy99] R. Buyya, editor. *High Performance Cluster Computing: Architectures and Systems.* Prentice_Hall, June 1999.

[Cai00] H. W. Cain, B. P. Miller, and B. J. Wylie. A callpgraph-based search strategy for automated performance diagnosis. In *Euro-Par 2000, Lecture Notes in Computer Science*, volume 1900, pages 108–122, 2000.

[Cap00] Franck Cappello, Olivier Richard, and Daniel Etiemble. Investigating the performance of two programming models for clusters of SMP PCs. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture*, pages 349–359, Toulouse, France, January 8–12, 2000. IEEE Computer Society TCCA.

[Cap99] Franck Cappello and Olivier Richard. Performance characteristics of a network of commodity multiprocessors for the NAS benchmarks using a hybrid memory model. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT '99)*, pages 108–116, Newport Beach, California, October 12–16, 1999. IEEE Computer Society Press.

[Ces02] Eduardo Cesar, Anna Morajko, Tomàs Margalef, et al. Dynamic performance tuning supported by program specification. *Scientific Programming*, 10(1):35–44, 2002.

[Ces99] Eduardo Cesar, Josep Jorba, Joan Sorribes, and Emilio Luque. Un modelo integrado para la especificación e implementación de aplicaciones distribuidas paralelas. In *X Jornadas de Paralelismo, Murcia, España*, 1999.

[Cha94] Satish Chandra, James R. Larus, and Anne Rogers. Where is time spent in message-passing and shared-memory programs? Technical Report TR-463-94, Princeton University, Computer Science Department, July 1994.

[Cha94a] B. Chapman, P. Mehrotra, and H. Zima. Extending HPF for advanced data-parallel applications. *IEEE Parallel and Distributed Technology*, 2(3):15–27, 1994.

[Cor95] T. Cortes, V. Pillet, J. Labarta, and S. Girona. PARAVER: A Tool to Visualize and Analyze Parallel Code. In Patrick Nixon, editor, *Proceedings of WoTUG-18: Transputer and occam Developments*, pages 17–31, 1995.

[Cou93] A.L. Couch. Locating performance problems in massively parallel executions. *Proceedings of the IEEE*, 81(8):1116–1125, 1993.

[Cra94] CRAY Research Inc. *Introducing the MPP Apprentice Tool*, cray manual in-2511 edition, 1994.

[Cro94] M. E. Crovella and T. J. LeBlanc. Parallel performance prediction using lost cycles analysis. In IEEE, editor, *Proceedings, Supercomputing '94: Washington, DC, November 14–18, 1994*, Supercomputing, pages 600–609. IEEE Computer Society Press, 1994.

[Cul99] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, inc., San Francisco, CA, 1999.

[Dau93] P. Dauphin. Knowledge bases in debugging parallel and distributed systems based on event traces. In *3rd ACM ONR workshop on Parallel and Distributed Debugging*, March 30 1993.

[Dau94] Peter Dauphin and Vassilis Mertsiotakis. MENTOR a model based event trace evaluation support system, January 24 1994.

[Del97] T. Delaitre, M. J. Zemerly, P. Vekariya, G. R. Ribeiro Justo, J. Bourgeois, F. Schinkmann, and Winter S. C. EDPEPPS: An environment

for the design and performance evaluation of portable parallel software. In *Workshop on Portable Software Tools for Parallel Applications, PSTPA'97*, Manchester, 1997.

[Del97a] T. Delaitre, George R. Ribeiro-Justo, François Spies, and S. C. Winter. A graphical toolset for simulation modelling of parallel systems. *Parallel Computing*, ??:??–??, 1997.

[Der02] Luiz DeRose and Felix Wolf. Catch - a call-graph based automatic tool for capture of hardware performance metrics for mpi and openmp applications. *Lecture Notes in Computer Science*, 2400:167–, January 2002.

[Dia05] Karl Dias, Mark Ramacher, Uri Shaft, Venkateshwaran Venkataramani, and Graham Wood-94. Automatic performance diagnosis and tuning in oracle. In *Proceedings of the CIDR 2005 conference*, 2005.

[Die98] Hank Dietz. Linux parallel processing HOWTO, January 06 1998.

[Dil95] E. Dillon, C. Gamboa, Dos Santos, J. Guyard, and Projet Resedas. Homogeneous and heterogeneous networks of workstations: Message passing overhead, June 25 1995.

[Don96] J. Dongarra, S. Otto, M. Snir, and D. Walker. A message passing standard for mpp and workstations. *Communications of the ACM*, 39(7):84–92, 1996.

[Dow93] Kevin Dowd. *High Performance Computing*. O'Reilly, 1993.

[Dun90] Ralph Duncan. A survey of parallel computer architectures. *IEEE Computer*, 23(2):5–16, February 1990.

[Esp00] Antonio Espinosa. *Automatic Performance Analysis of Parallel Programs.* PhD thesis, Universidad Autonoma de Barcelona, Julio 2000.

[Esp00a] Antonio Espinosa, Tomas Margalef, and Emilio Luque. Integrating automatic techniques in a performance analysis session (research note). *Lecture Notes in Computer Science*, 1900:173–177, 2000.

[Esp00b] Antonio Espinosa, Tomas Margalef, and Emilio Luque. Automatic performance analysis of master/worker pvm applications with kpi. *Lecture Notes in Computer Science*, 1908:47–, January 2000.

[Esp97] Antonio Espinosa, Tomas Margalef, and Emilio Luque. Knowledge-based automatic performance analysis of parallel programs. In *PARCO*, pages 697–700, 1997.

[Esp98] A. Espinosa, T. Margalef, and E. Luque. Automatic detection of pvm program performance problems. *Lecture Notes in Computer Science*, 1497:19–, January 1998.

[Fah00] T. Fahringer, M. Gerndt, G. Riley, and J. L. Träff. Specification of performance problems in MPI programs with ASL. In *2000 International Conference on Parallel Processing (ICPP'00)*, pages 51–60, Washington - Brussels - Tokyo, August 2000. IEEE.

[Fah01] Thomas Fahringer and Clovis Seragiotto Junior. Modeling and detecting performance problems for distributed and parallel programs with javaPSL. In *SC'2001 Conference CD*, Denver, November 2001. ACM SIGARCH/IEEE. University of Vienna.

[Fah01a] Thomas Fahringer, Michael Gerndt, Bernd Mohr, Felix Wolf, Graham Riley, Interner Bericht, and Jesper Larsson Traff. Knowledge specification for automatic performance analysis (apart technical report). Technical report, APART 2 Project, January 30 2001.

[Fah02] Thomas Fahringer, Clovis Seragiotto, and Clovis Seragiotto. Automatic search for performance problems in parallel and distributed programs by using multi-experiment analysis. *Lecture Notes in Computer Science*, 2552:151–162, January 2002.

[For95] Message Passing Interface Forum. Mpi: A message-passing interface standard. `http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html`, 1995.

[For97] Message Passing Interface Forum. Mpi-2: Extensions to the message-passing interface. `http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html`, 1997.

[Fos95] I Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 95.

[Fos99] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a Future Computing Infrastructure*. MORGAN-KAUFMANN, 1999.

[Fox98] Mark Robert Fox. Event-predicate detection in the monitoring of distributed applications. Master's thesis, University of Waterloo, Ontario, Canada, 1998.

[Fre02] Felix Freitag, Jordi Caubet, and Jesus Labarta. On the scalability of tracing mechanisms. In Burkhard Monien and Rainer Feldmann, editors, *Proceedings of the 8th International Euro-Par Conference on Parallel Processing, EURO-PAR'2002 (Paderborn, Germany, August 27-30, 2002)*, volume 2400 of *LNCS*, pages 97–104. Springer-Verlag, 2002.

[Fri97] M. Frigo and S. G. Johnson. The fastest fourier transform in the west. Technical Report MIT-LCS-TR-728, MIT, 1997.

[Gei94] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidyalingam S. Sunderam. *PVM: Parallel Virtual Machine A Users' Guide and Tutorial for Network Parallel Computing*. Scientific and Engineering Computation Series. MIT Press, Cambridge, MA, 1994.

[Ger02] Michael Gerndt. Specification of performance properties of hybrid programs on hitachi sr8000. Technical report, Institut fur Informatik. Munchen, Feb 2002.

[Ger04] Michael Gerndt, Bernd Mohr, and Jesper Larsson Traff. Evaluating openmp performance analysis tools with the apart test suite. *Lecture Notes in Computer Science*, 3149:155–162, January 2004.

[Ger97] M. Gerndt and A. Krumme. A rule-based approach for automatic bottleneck detection inprograms on shared virtual memory systems. In *High-Level Programming Models and Supportive Environments, 1997. Proceedings., Second International Workshop on*, pages 93–101, 1997.

[Gio99] M. Giordano, M.M. Furnari, and F. Vitobello. Pvm parameter tuning to improve communication in distributedapplications. In *EUROMICRO Conference, 1999. Proceedings. 25th*, volume 2, pages 438–445 vol.2, 1999.

[Gir00] Sergi Girona and Jesús Labarta. Sensitivity of performance prediction of message passing programs. *The Journal of Supercomputing*, 17(3):291–298, November 2000.

[Gor00] S. Gorlatch. Toward formally-based design of message passing programs. *Software Engineering, IEEE Transactions on*, 26(3):276–288, 2000.

[Gro96] William Gropp, Ewing Lusk Anthony Skjellum, and Nathan Doss. A high-performance, portable implementation of the MPI message passing interface standard, December 26 1996.

[Gro99] William Gropp and Ewing Lusk. Reproducible measurements of mpi performance characteristics. *Lecture Notes in Computer Science*, 1697:11–, January 1999.

[Gu98] Weiming Gu, Greg Eisenhauer, Karsten Schwan, and Jeffrey S. Vetter. Falcon: On-line monitoring for steering parallel programs. *Concurrency and Computation: Practice and Experience (Concurrency - Practice and Experience)*, ??:??–??, 1998.

[Gus88] John L. Gustafson. Reevaluating amdahl's law. *Communications of the ACM (CACM)*, 31(5):532–533, May 1988.

[Gus88a] John L. Gustafson. The scaled-sized model: A revision of amdahl's law. In *Proceeding Supercomputing '88: Technology Assessment, Industrial Supercomputer Outlooks, European Supercomputing Accomplishments, and Performance & Computations*, volume 2, pages 130–133, St. Petersberg, FL, 1988. Third International Conference on Supercomputing (ICS '88).

[Han93] Per Brinch Hansen. Model programs for computational science: A programming methodology for multicomputers. *Concurrency: Practice & Experience*, 5(5):407–423, August 1993.

[Hat91] Philip J. Hatcher and Michael J. Quinn. *Data–Parallel Programming on MIMD Computers*. The MIT Press, Cambridge, MA, 1991.

[Hea95] M.T. Heath, A.D. Malony, and D.T. Rover. The visual display of parallel performance data. *Computer*, 28(11):21–28, 1995.

[Hel89] D. P. Helmbold and C. E. McDowell. Modeling speedup greater than n. In *International Confrence on Parallel Processing, Vol.3: Algorithms and Applications (ICPP '89)*, pages 219–225, Pennsylvania, USA, August 1989. The Pennsylvania State University.

[Hel96] Joseph Hellerstein. An approach to selecting metrics for detecting performance problems in information systems. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer SYstems*, volume 24,1 of *ACM SIGMETRICS Performance Evaluation Review*, pages 266–267, New York, May 23–26 1996. ACM Press.

[Hey00] Tony Hey and David Lancaster. The development of parkbench and performance prediction. *International Journal of High Performance Computing Applications*, 14(3):205–215, 2000.

[Hey97] Tony Hey, Alistair Dunlop, and Emilio Hernández. Realistic parallel performance estimation. *Parallel Computing*, 23(1–2):5–21, April 1997.

[Hil86] W. Daniel Hillis and Guy L. Steele. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, December 1986.

[Hol02] J.K. Hollingsworth and B. Buck. *DyninstAPI Programmer's Guide. Release 3.0*. University of Maryland, Jan 2002.

[Hol93] Jeffery K. Hollingsworth and Barton P. Miller. Dynamic control of performance monitoring on large scale parallel systems. In *Conference Proceedings, 1993 International Conference on Supercomputing*, pages 185–194, Tokyo, July 20–22, 1993. ACM SIGARCH.

[Hol98] J.K. Hollingsworth. Critical path profiling of message passing and shared-memoryprograms. *Parallel and Distributed Systems, IEEE Transactions on*, 9(10):1029–1040, 1998.

[Hor01] Mihai Horoi and Richard J. Enbody. Using AMDAHL's Law as a Metric to Drive Code Parallelization: Two Case Studies. *International Journal of High Performance Computing Applications*, 15(1):75–80, 2001.

[Hub01] Simon Huband and Chris McDonald. A preliminary topological debugger for MPI programs. In *International Symposium on Cluster Computing and the Grid*, May 15 2001.

[Hub99] S. Huband and C. McDonald. Debugging parallel programs using incomplete information. In *Cluster Computing, 1999. Proceedings. 1st IEEE Computer Society International Workshop on*, pages 278–286, 1999.

[Irv96] R. Bruce Irvin and Barton P. Miller. Mapping performance data for high-level and data views of parallel program performance. In *International Conference on Supercomputing (ICS'96)*, Phil., PA (Federated Comp. Res. Conf.), May 1996. ACM. TR '95Published as International Conference on Supercomputing (ICS'96), volume ?, number ?

[Iva04] Vicente J. Ivars. Monitor de aplicaciones mpich basado en dyninst. Master's thesis, Universidad Autonoma de Barcelona, sep 2004.

[Jai91] Raj Jain. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, Inc., 1991.

[Jia04] Weihang Jiang, Jiuxing Liu, Hyun-Wook Jin, D.K. Panda, W. Gropp, and R. Thakur. High performance mpi-2 one-sided communication over infiniband. In *Cluster Computing and the Grid, 2004. CCGrid 2004. IEEE International Symposium on*, pages 531–538, 2004.

[Jor01] Josep Jorba, Tomas Margalef, and Emilio Luque. Simulation of forest fire propagation on parallel and distributed pvm platforms. *Lecture Notes in Computer Science*, 2131:386–, January 2001.

[Jor02] Josep Jorba. Static performance analysis: Kappapi 2 state of art (apart project 2002 meeting notes). `http://www.fz-juelich.de/apart-2/meet5/kpi2.ppt`, June 2002.

[Jor05] Josep Jorba, Tomas Margalef, and Emilio Luque. Automatic performnace analysis of message passing applications using the kappapi 2 tool. *Lecture Notes in Computer Science*, 3666:293–300, September 2005.

[Jor05a] Josep Jorba, Tomas Margalef, and Emilio Luque. Performance analysis of parallel applications with kappapi 2. In *Procedings of PARCO 2005 Conference*, 2005.

[Jor98] Josep Jorba, Tomas Margalef, Emilio Luque, J. Andre, and D. Viegas. Application of parallel processing to the simulation of forest fire propagation. In *Proceedings of the III International Conference on Forest Fire Research*, volume II, pages 891–900, Coimbra, Portugal, 1998.

[Jor99] Josep Jorba, Tomas Margalef, and Emilio Luque. Simulación paralela de propagación de incendios forestales. In *Proceedings de las X Jornadas de Paralelismo*, 1999.

[Kar00] N.T. Karonis, B.R. de Supinski, I. Foster, W. Gropp, E. Lusk, and J. Bresnahan. Exploiting hierarchy in parallel computer networks to optimizecollective operation performance. In *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, pages 377–384, 2000.

[Kar90] Alan H. Karp and Horace P. Flatt. Measuring parallel processor performance. *Communications of the ACM*, 33(5):539–543, May 1990.

[Kar97] Karen L. Karavanic and Barton P. Miller. Experiment management support for performance tuning. In ACM, editor, *SC'97: High Performance Networking and Computing: Proceedings of the 1997 ACM/ IEEE SC97 Conference: November 15–21, 1997, San Jose, California, USA.*, pages ??–??, pub-ACM:adr and pub-IEEE:adr, 1997. pub-ACM and pub-IEEE.

[Klu05] Michael Kluge, Andreas Knupfer, and Wolfgang E. Nagel. Statistical methods for automatic performance bottleneck detection in mpi based programs. *Lecture Notes in Computer Science*, 3514:330–337, January 2005.

[Klu05a] Michael Kluge, Andreas Knupfer, and Wolfgang E. Nagel. Knowledge based automatic scalability analysis and extrapolation for mpi programs. *Lecture Notes in Computer Science*, 3648:176–184, August 2005.

[Knu04] Andreas Knupfer, Dieter Kranzlmuller, and Wolfgang E. Nagel. Detection of collective mpi operation patterns. *Lecture Notes in Computer Science*, 3241:259–267, January 2004.

[Knu05] Andreas Knupfer and Wolfgang E. Nagel. New algorithms for performance trace analysis based on compressed complete call graphs. *Lecture Notes in Computer Science*, 3515:116–123, January 2005.

[Knu05a] A. Knupfer and W.E. Nagel. Construction and compression of complete call graphs for post-mortem program trace analysis. In *Parallel Processing, 2005. ICPP 2005. International Conference on*, pages 165–172, 2005.

[Knu05b] A. Knupfer, H. Brunst, and W.E. Nagel. High performance event trace visualization. In *Parallel, Distributed and Network-Based Processing, 2005. PDP 2005. 13th Euromicro Conference on*, pages 258–263, 2005.

[Koh96] J. A. Kohl and G. A. Geist. The PVM 3.4 tracing facility and XPVM 1.1. In Hesham El-Rewini and Bruce D. Shriver, editors, *Proceedings of the Twenty-Ninth Hawaii International Conference on System Sciences (HICSS-29): Wailea, HI, USA, 3–6 January 1996*, volume 1, pages 290–299, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1996. IEEE Computer Society Press.

[Kra00] Dieter Kranzlmuller. *Event Graph Analysis for Debugging Massively Parallel Programs*. PhD thesis, Johennes Kepler Universitat Linz, Departement for Graphics and Parallel Procesing, Austria, 2000.

[Kri96] Sanjeev Krishnan and Laxmikant V. Kale. Automating parallel runtime optimizations using post-mortem analysis. In *Conference Proceedings, 1996 International Conference on Supercomputing*, pages 221–228, Philadelpha, Pennsylvania, May 25–28, 1996. ACM SIGARCH.

[Kri96a] Sanjeev Krishnan. *Automating Runtime Optimizations For Parallel Object-Oriented Programming*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, jun 1996.

[Kuh04] M. Kuhnemann, T. Rauber, and G. Runger. A source code analyzer for performance prediction. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pages 262–, 2004.

[Kum94] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin/Cummings, Redwood City, CA, 1994.

[Kun93] Thomas Kunz. Event abstraction: Some definitions and theorems. Technical report, Fachbereich Informatik and Technische Hochschule Darmstadt, February 21 1993.

[Kun94] Thomas Kunz. An event abstraction tool: Theory, design, and results. Technical report, Fachbereich Informatik and Technische Hochschule Darmstadt, February 21 1994.

[Lan96] Pat Langley. *Elements of Machine Learning*. Morgan Kaufmann, 1996.

[Li96] K.C. Li and K. Zhang. Tuning parallel program through automatic program analysis. In *Parallel Architectures, Algorithms, and Networks, 1996. Proceedings. Second International Symposium on*, pages 330–333, 1996.

[Mai95] Eric Maillet. *TAPEPVM an efficient performance monitor for PVM applications - User guide*. LMC-IMAG, Grenoble, France, June 1995.

[Mal94] A. D. Malony, V. Mertsiotakis, and A. Quick. Automatic scalability analysis of parallel programs based on modelling techniques. *Lecture Notes in Computer Science*, 794:139–158, 1994.

[Mar04] Tomas Margalef, Josep Jorba, Oleg Morajko, Anna Morajko, and Emilio Luque. *Performance Analysis and Grid Computing*, chapter Different approaches to automatic performance analysis of distributed applications, pages 3–19. Kluwer Academic Publishers, Norwell, MA, USA, 2004.

[Mar99] Benjamino Di Martino, Antonino Mazzeo, Nicola Mazzocca, and Umberto Villano. Restructuring parallel programs by transformation of point-to-point interactions into collective communication. In *Proceedings: Seventh International Workshop on Program Comprehension*, pages 84–91. IEEE Computer Society Press, 1999.

[McB94] Oliver A. McBryan. An overview of message passing environments. *Parallel Computing*, 1994.

[Mei95] Wagner Meira. Modeling performance of parallel programs. Technical Report 589, University of Rochester, Computer science Departement, jun 1995.

[Mid04] E.T. Midorikawa, H.M. de Oliveira, and J.M. Laine. Pempis: a new methodology for modeling and prediction of mpi programs performance. In *Computer Architecture and High Performance Computing, 2004. SBAC-PAD 2004. 16th Symposium on*, pages 246–253, 2004.

[Mil95] B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam, and T.Ñewhall. The paradyn parallel performance measurement tool. *Computer*, 28(11):37–46, 1995.

[Mit97] Tom Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[Moh01] Bernd Mohr, Allen D. Malony, Sameer Shende, and Felix Wolf. Design and prototype of a performance tool interface for openmp. *The Journal of Supercomputing*, 23(1):105–128, May 2001.

[Moh03] B. Mohr and J.L. Traff. Initial design of a test suite for automatic performance analysis tools. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 10 pp.–, 2003.

[Moh03a] Bernd Mohr and Felix Wolf. Kojak a tool set for automatic performance analysis of parallel programs. *Lecture Notes in Computer Science*, 2790:1301–1304, January 2003.

[Moh93] Bernd Mohr. Standardization of event traces considered harmful or is an implementation of object-independent event trace monitoring and analysis systems possible? In Jack J. Dongarra and Bernard Tourancheau, editors, *Environments and Tools for Parallel Scientific Computing*, volume 6 of *Advances in Parallel Computing*, pages 103–124. Elsevier Science Publishers B. V., Sara Burgerhartstraat 25, P.O. Box 211, 1000 AE Amsterdam, The Netherlands, 1993.

[Moh94] B. Mohr, D. Brown, and A. Malony. TAU: A portable parallel program analysis environment for pC++. *Lecture Notes in Computer Science*, 854:29–??, 1994.

[Mol93] D. I. Moldovan. *Parallel Processing - From Applications to Systems*. Morgan Kaufmann, San Mateo, 1993.

[Moo01] Shirley Moore, David Cronk, Kevin London, and Jack Dongarra. Review of performance analysis tools for MPI parallel programs. *Lecture Notes in Computer Science*, 2131:241–??, 2001.

[Mor03] Anna Morajko, Oleg Morajko, Josep Jorba, Tomas Margalef, and Emilio Luque. Automatic performance analysis and dynamic tuning of distributed applications. *Parallel Processing Letters*, 13(2):169–187, 2003.

[Mor03a] Anna Morajko, Oleg Morajko, Josep Jorba, Tomas Margalef, and Emilio Luque. Dynamic performance tuning of distributed programming libraries. *Lecture Notes in Computer Science*, 2660:191–200, January 2003.

[Muk01] Nandini Mukherjee, Graham D. Riley, and John R. Gurd. A preliminary evaluation of FINESSE, a feedback-guided performance enhancement system. *Lecture Notes in Computer Science*, 1900:75–??, 2001.

[Muk99] N. Mukherjee, G. Riley, and J. Gurd. Finesse: A prototype feedback-guided performance enhancement system. In Bob Werner, editor, *Proceedings of the Euromicro Workshop on Parallel and Distributed Processing*, pages 101–109, Los Alamitos, CA, January 19–21 1999. IEEE Computer Society.

[Nag96] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12(1):69–80, January 1996.

[Net92] R. H. B. Netzer and B. P. Miller. Optimal tracing and replay for debugging message-passing parallel programs. In *Proceedings of the Conference on Supercomputing*, pages 502–511, Los Alamitos, CA, USA, November 1992. IEEE Computer Society Press.

[Pan99] C. M. Pancake. Applying human factors to the design of performance tools. *Lecture Notes of Computer Science*, 1685:44–60, 1999.

[Par98] P. J. Parsons and F. A. Rabhi. Generating parallel programs from skeleton based specifications. *Journal of Systems Architecture*, 45(4):261–283, December 1998.

[Pje05] J. Pjesivac-Grbovic, T. Angskun, G. Bosilca, G.E. Fagg, E. Gabriel, and J.J. Dongarra. Performance analysis of mpi collective operations. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 272a–272a, 2005.

[Pll05] S. Pllana and T. Fahringer. Performance prophet: A performance modeling and prediction tool for parallel and distributed programs. In *Parallel Processing, 2005. ICPP 2005 Workshops. International Conference Workshops on*, pages 509–516, 2005.

[Pra00] S. Prakash, E. Deelman, and R. Bagrodia. Asynchronous parallel simulation of parallel programs. *Software Engineering, IEEE Transactions on*, 26(5):385–400, 2000.

[Pro05] The Open MPI Project. Open mpi: Open source high performance computing. `http://www.open-mpi.org`, 2005.

[Psa04] K. Psarris and K. Kyriakopoulos. An experimental evaluation of data dependence analysis techniques. *Parallel and Distributed Systems, IEEE Transactions on*, 15(3):196–213, 2004.

[Ran02] William T. Rankin. Dpmta - a distributed implementation of the parallel multipole tree algorithm - version 3.1. Technical report, Department of Electrical Enginnering, Duke University, 2002.

[Rau00] T. Rauber and G. Runger. Modelling the runtime of scientific programs on parallel computers. In *Parallel Processing, 2000. Proceedings. 2000 International Workshops on*, pages 307–314, 2000.

[Rau97] Thomas Rauber and Gudula Runger. PVM and MPI communication operations on the IBM SP2: Modeling and comparison. Technical report, Fachbereich Informatik, November 20 1997.

[Ree93] D. A. Reed, R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. W. Schwartz, and L. F. Tavera. Scalable performance analysis: The pablo performance analysis environment. In A. Skjellum, editor, *Proceedings of the Scalable Parallel Libraries Conference*, pages 104–113. IEEE Computer Society, 1993.

[Rib01] Randy L. Ribler, Huseyin Simitci, and Daniel A. Reed. The Autopilot performance-directed adaptive control system. *Future Generation Computer Systems*, 18(1):175–187, September 2001.

[Ros98] L. De Rose, Y. Zhang, and D. A. Reed. SvPablo: A multi-language performance analysis system. *Lecture Notes in Computer Science*, 1469:352–??, 1998.

[Ser03] C. Seragiotto, T. Fahringer, M. Geissler, G. Madsen, and H. Moritsch. On using aksum for semi-automatically searching of performance problems in parallel and distributed programs. In *Parallel, Distributed and Network-Based Processing, 2003. Proceedings. Eleventh Euromicro Conference on*, pages 385–392, 2003.

[Sin96] Amitabh Sinha and Laxmikant V. Kalé. Towards automatic performance analysis. In *International Conference on Parallel Processing (ICPP)*, 1996.

[Sin97] Ajit Singh and Vincent Van Dongen. An integrated performance analysis tool for SPMD data-parallel programs. *Parallel Computing*, 23:1089–1112, 1997.

[Sna02] Allan Snavely, Laura Carrington, Nicole Wolter, and The San. A framework for performance modeling and prediction, August 30 2002.

[Ste02] Aad van der Steen and Jack Dongarra. Overview of high performance computers. In *Handbook of Massive Data Sets*, pages ??–?? Kluwer Academic Publishers Group, Norwell, MA, USA, and Dordrecht, The Netherlands, 2002.

[Ste96] Aad J. van der Steen and Jack J. Dongarra. Overview of recent supercomputers. Technical Report UT-CS-96-325, Department of Computer Science, University of Tennessee, April 1996.

[Sum00] K.L. Summers, J. Greenfield, and B.T. Smith. A survey of parallel program performance evaluation techniques using visualization and virtual reality. In *Aerospace Conference Proceedings, 2000 IEEE*, volume 4, pages 457–464, 2000.

[Sun02] Xian-He Sun, Thomas Fahringer, and Mario Pantano. SCALA: A Performance System For Scalable Computing. *International Journal of High Performance Computing Applications*, 16(4):357–370, 2002.

[Sup99] B.R. de Supinski and N.T. Karonis. Accurately measuring mpi broadcasts in a computational grid. In *High Performance Distributed Computing, 1999. Proceedings. The Eighth International Symposium on*, pages 29–37, 1999.

[Top05] TOP500.Org. Top500 supercomputer sites. `http://www.top500.org/`, 2005.

[Tru01] Hong-Linh Truong, Thomas Fahringer Georg Madsen, Allen D. Malony, Hans Moritsch, and Sameer Shende. On using SCALEA for performance analysis of distributed and parallel programs. In ACM, editor, *SC2001: High Performance Networking and Computing. Denver, CO, November 10–16, 2001*, pages ??–??, New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 2001. ACM Press and IEEE Computer Society Press.

[Tse02] Y. Tseng and R.F. DeMara. Communication pattern based methodology for performance analysis of termination detection schemes. In *Parallel and Distributed Systems, 2002. Proceedings. Ninth International Conference on*, pages 535–541, 2002.

[Uni05] Indiana University. Lam/mpi parallel computing. `http://www.lam-mpi.org`, 2005.

[Vad01] Sathish S. Vadhiyar, Graham E. Fagg, and Jack J. Dongarra. Towards an accurate model for collective communications. *Lecture Notes in Computer Science*, 2073:41–??, 2001.

[Vad04] Sathish S. Vadhiyar, Graham E. Fagg, and Jack J. Dongarra. Towards an accurate model for collective communications. *International Journal of High Performance Computing Applications*, 18(1):159–167, 2004.

[Vet00] Jeffrey Vetter. Performance analysis of distributed applications using automatic classification of communication inefficiencies. In *ICS '00: Proceedings of the 14th international conference on Supercomputing*, pages 245–254, New York, NY, USA, 2000. ACM Press.

[Vud04] Richard Vuduc, James W. Demmel, and Jeff A. Bilmes. Statistical Models for Empirical Search-Based Performance Tuning. *International Journal of High Performance Computing Applications*, 18(1):65–94, 2004.

[Wil99] Barry Wilkinson and Michael Allen. *Parallel Programming*. Prentice Hall, 1999.

[Wol00] Felix Wolf, Bernd Mohr, Forschungszentrum Jlich Gmbh, and Interner Bericht. Specifying performance properties using compound runtime events. Technical report, Central Institute for Applied Mathematics, Research Centre Juelich, August 24 2000.

[Wol01] Felix Wolf and Bernd Mohr. Automatic performance analysis of MPI applications based on event traces. *Lecture Notes in Computer Science*, 1900:123–??, 2001.

[Wol02] Felix Wolf, Forschungszentrum Jlich, H. Bischof, Hochschule Aachen, Nic Series, Ph. D, Printer Graphische Betriebe, Rheinisch westfalischen Technischen, Universitatsprofessor Christian, and Volume Isbn. *Automatic Performance Analysis on Parallel Computers with SMP Nodes*. PhD thesis, Research Centre Jlich; Address: 52425 Jlich, Germany, February 12 2002.

[Wol03] F. Wolf and B. Mohr. Automatic performance analysis of hybrid mpi/openmp applications. In *Parallel, Distributed and Network-Based Processing, 2003. Proceedings. Eleventh Euromicro Conference on*, pages 13–22, 2003.

[Wol04] Felix Wolf, Bernd Mohr, Jack Dongarra, and Shirley Moore. Efficient pattern search in large traces through successive refinement. *Lecture Notes in Computer Science*, 3149:47–54, January 2004.

[Wol99] F. Wolf and B. Mohr. EARL — A programmable and extensible toolkit for analyzing event traces of message passing programs. *Lecture Notes in Computer Science*, 1593:503–??, 1999.

[Wor01] P. H. Worley. MPI performance evaluation and characterization using a compact application benchmark code, May 04 2001.

[Wu00] Xingfu Wu, Valerie E. Taylor, Jonathan Geisler, Xin Li, Zhiling Lan, Rick L. Stevens, Mark Hereld, and Ivan R. Judson. Prophesy: An infrastructure for analyzing and modeling the performance of parallel and distributed applications. In *IEEE International Symposium on High Performance Distributed Computing (HPDC)*, 2000.

[Wu01] Xingfu Wu, Valerie Taylor, and Rick Stevens. Design and implementation of prophesy automatic instrumentation and data entry system. In *Proc. of the 13th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS2001)*, Anaheim, CA, aug 2001.

[Wu01a] Xingfu Wu, Valerie Taylor, Jonathan Geisler, Xin Li, Zhiling Lan, Rick Stevens, Mark Hereld, and Ivan R. Judson. Design and development of prophesy performance database for distributed scientific applications. In *Proc. the 10th SIAM Conference on Parallel Processing for Scientific Computing*, Virginia, march 2001.

[Yan96] Jerry C. Yan and Sekhar R. Sarukkai. Analyzing parallel program performance using normalized performance indices and trace transformation techniques. *Parallel Computing*, 22(9):1215–1237, 1996.

[Zak99] Omer Zaki, Ewing Lusk, William Gropp, and Deborah Swider. Toward Scalable Performance Visualization with Jumpshot. *International Journal of High Performance Computing Applications*, 13(3):277–288, 1999.

[Zav00] A. Zavanella and A. Milazzo. Predictability of Bulk Synchronous Programs Using MPI. In *Proceedings of Euromicro Parallel and Distributed Processing 2000*, pages 118–123. IEEE, 2000.

[Zhe00] Qilong Zheng, Guoliang Cheng, and Liusheng Huang. Optimal record and replay for debugging of nondeterministic mpi/pvmprograms. In *High Performance Computing in the Asia-Pacific Region, 2000. Proceedings. The Fourth International Conference/Exhibition on*, volume 1, pages 473–475 vol.1, 2000.