Análisis automático del rendimiento de ejecución de los programas paralelos.

Antonio Espinosa, Tomàs Margalef, Emilio Luque.
Universitat Autònoma de Barcelona. Departamento de informática.
08193 Bellaterra, (Barcelona)
{a.espinosa, t.margalef, iinfd}@cc.uab.es, telefono: +34 93 581 19 90

Resumen:
La programación paralela más tradicional obliga al programador, después de haber diseñado la aplicación, a analizar el rendimiento de la aplicación que se acaba de diseñar. El programador debe analizar una enorme cantidad de información obtenida de la ejecución de un programa. En este artículo, se propone una herramienta de análisis automático que permite a los programadores evitar esa tarea difícil. La herramienta se centra en la búsqueda de intervalos de ineficiencia a lo largo del tiempo de ejecución de la aplicación. Los problemas de rendimiento son analizados en detalle, en busca de posibles causas de la ineficiencia y sus soluciones.

Palabras clave:

Diseño de programas paralelos, análisis de rendimiento, programación paralela.

1. Introducción.

El buen rendimiento de las aplicaciones paralelas es su razón de ser [Pan 95]. Todos los esfuerzos en diseñar y construir una aplicación paralela pueden verse frustrados si el rendimiento de la aplicación no cumple los requerimientos definidos para el programa. De esta forma, el análisis del rendimiento y su mejora se convierten en fases vitales para el desarrollo de una aplicación paralela.

El ciclo más clásico para analizar el rendimiento de una aplicación paralela se divide en una serie de etapas: diseño de la aplicación paralela; ejecución de la aplicación, incluyendo la monitorización necesaria para generar un fichero de traza, y, finalmente, el análisis de estos ficheros de traza utilizando una herramienta de visualización.

La gran mayoría de herramientas de visualización como Paragraph [Hea 91] y Pablo [Ree 93] ofrecen a los programadores la posibilidad de ver los datos de ejecución en una larga serie de ventanas gráficas que representan diferentes aspectos de la ejecución. A partir de estas ventanas gráficas, los programadores deben extraer la información necesaria para responder preguntas como: ¿cuál es el rendimiento global de la aplicación? ¿Dónde decae el rendimiento durante la ejecución? ¿Cuáles son las causas de esta caída del rendimiento? ¿Cómo se puede mejorar ese rendimiento?.

Cuando el programador trata de resolver las preguntas anteriores, debe enfrentarse a problemas como el comprender la gran cantidad de información gráfica que se abre al iniciar una sesión de análisis de rendimiento. Para ello, debe seleccionar diferentes vistas de datos que explican aspectos importantes del rendimiento. Además, debe saber relacionar y entender la información que se le ofrece en las ventanas de forma que pueda deducir qué problemas aparecen en la ejecución de la aplicación. Finalmente, deberá relacionar la información de bajo nivel obtenida de las representaciones gráficas con los procedimientos y las estructuras de datos de la aplicación que el programador ya conoce.

De esta forma, los programadores necesitan un alto nivel de experiencia para ser capaces de obtener conclusiones respecto al comportamiento de su aplicación al utilizar este camino de visualización del rendimiento. Además, también deben tener un profundo conocimiento del sistema paralelo ya que el análisis de muchos aspectos del rendimiento debe considerar aspectos de la arquitectura como la topología del sistema o la red de interconexión.

En este artículo, presentamos una herramienta llamada KAPPA-PI (de "Knowledge-based Automatic Parallel Program Analyser for Performance Improvement) que realiza un análisis post-mortem del fichero de traza y proporciona algunas sugerencias acerca del comportamiento del programa. Utilizando esta herramienta, los programadores pueden llegar a ahorrarse una considerable cantidad de esfuerzo y tiempo de análisis.

Existen otras aproximaciones anteriores a esta herramienta con objetivos similares:
- Paradyn [Hol 93]. Esta herramienta se centra en la búsqueda de problemas de rendimiento mediante un módulo llamado "Performance Consultant". Este módulo pretende minimizar la instrumentación sobre el sistema medido realizándola de forma dinámica al tiempo que se ejecuta la aplicación.

- Otra aproximación para resolver el problema de analizar el rendimiento de un programa paralelo es la mostrada en [Cro 94] y [Mei 93]. La solución propuesta es construir una representación abstracta del programa con la ayuda de un modelo del sistema paralelo. Esta representación del programa se analiza con la intención de predecir algunos aspectos del comportamiento del programa. El problema más importante de esta aproximación es el propio modelo, ya que si se hace desde un nivel alto, algunos aspectos de su rendimiento pueden no llegar a ser considerados, al quedar fuera de la representación abstracta.

- El rendimiento del programa también puede ser medido con un pre-compilador, como en los programas Fortran (P3T, [Fah 93]). El problema está en la imposibilidad de aplicar esta solución a todos los programas paralelos, habida cuenta de las situaciones en las que el programador expresa un comportamiento dinámico sin estructurar entre los procesos de la aplicación.

La herramienta KAPPA-PI está implementada actualmente para analizar los problemas de rendimiento de las aplicaciones que utilizan PVM como librería de paso de mensajes. KAPPA-PI, tal como refleja la figura 1, basa su búsqueda de problemas de rendimiento en el conocimiento que tiene de sus causas, clasificando aquellos intervalos con un rendimiento más bajo con la ayuda de una "base de conocimiento" de las causas de los problemas de rendimiento. El proceso de análisis, identifica los problemas más importantes que afectan al rendimiento de la aplicación y crea recomendaciones para que el usuario mejore el comportamiento de la aplicación. La "base de conocimiento" de los problemas de los programas paralelos está construida de forma que se puede adaptar fácilmente a la incorporación de nuevos problemas y a distintos tipos de modelos de programación.

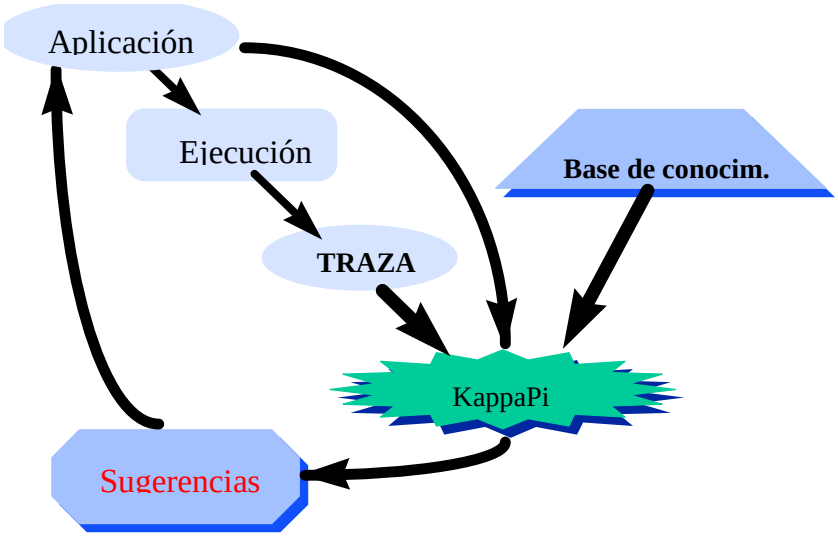

Figura 1: metodología de funcionamiento de la herramienta KAPPA-PI.

En la sección 2 se describe la metodología del análisis brevemente, explicando los principios de su operación y los pasos para procesar los datos de rendimiento que sirven de entrada a la herramienta. En la sección 3 se presenta la clasificación de problemas y sus causas. Esta sección se centrará en cómo se detectan los problemas y cómo se construyen las soluciones que se presentan al usuario para solventarlos. La sección 4 introduce un ejemplo de funcionamiento de la herramienta. Finalmente, la sección 5 expone las conclusiones y el trabajo futuro en el desarrollo de la herramienta.

2. Metodología de operación .

En esta sección, vamos a introducir los principios de la metodología para realizar un análisis del rendimiento de una aplicación. Esta metodología puede ser aplicada a diferentes modelos de programación, distintos acercamientos a la monitorización de los datos de ejecución o plataformas distintas sin grandes cambios en los principios generales del análisis.

El análisis automático del rendimiento, cuyas fases se expresan en la figura 2, se basa en la lectura y el procesamiento de los ficheros de traza obtenidos de la ejecución de la aplicación. Este análisis se divide en dos grandes etapas. La primera, la fase de detección, se basa en el análisis del fichero de traza en busca de ineficiencias en la ejecución, ordenándolas según su influencia en la ejecución. La segunda, la clasificación del problema, busca una causa para los problemas de rendimiento más importantes.

En los siguientes puntos, vamos a describir en detalle las fases del análisis.

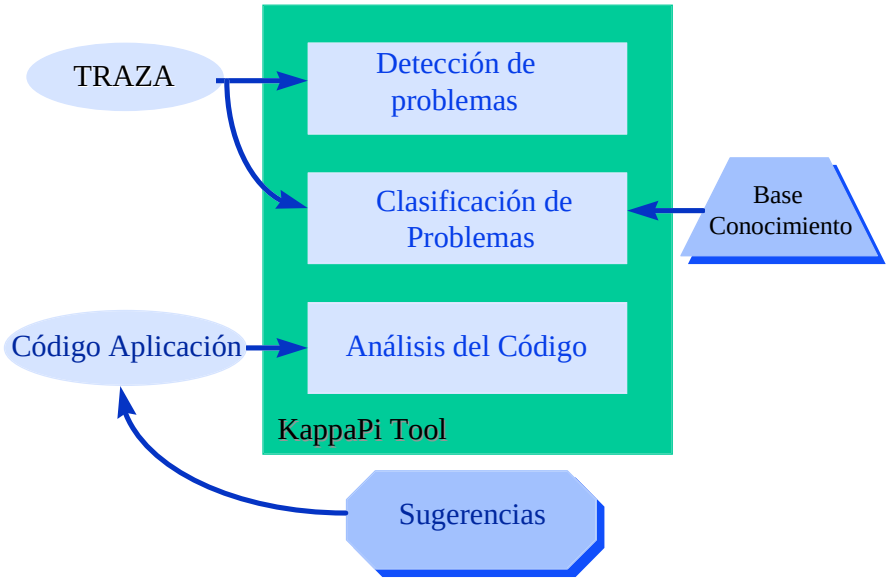

Figura 2: etapas en el análisis de rendimiento.

2.1. Monitorización.

El análisis automático del rendimiento se basa en el procesamiento de los ficheros de traza de ejecución obtenidos mediante el uso de la herramienta de monitorización TapePVM [Mai 95]. Uno de los problemas de considerar ficheros de traza como representación de la ejecución de un programa es la representatividad de una única traza. Una única traza no tiene por qué representar la ejecución completa de una aplicación debido a que puede mostrar diferentes comportamientos según los datos de entrada. El análisis del rendimiento de una aplicación debería ser considerado una vez la aplicación muestra una serie de resultados estables al variar los datos de entrada.

2.2. Análisis del fichero de traza

El objetivo de la fase de análisis del fichero de traza es encontrar problemas de ineficiencia de la aplicación y descubrir las causas de esos problemas de eficiencia encontrados.

El análisis de la traza se realiza en una serie de etapas, desde la clasificación general del comportamiento de la aplicación a la clasificación de las causas de los problemas encontrados.

1. El primer paso en el proceso de análisis es una visión general del rendimiento de la aplicación. Debido a que los ficheros de traza son normalmente muy extensos, el análisis filtra inicialmente los eventos de la traza, considerando sólo los tiempos de ocupación de los procesadores. En otras palabras, el fichero de traza es analizado para obtener una expresión dinámica del número de procesadores activos en el sistema a lo largo de la ejecución de la aplicación. Esta simplificación permite sólo considerar un subconjunto de los eventos de la traza, reduciendo así el peso del procesamiento de la traza.
2. A partir de los valores de eficiencia obtenidos, el siguiente paso es comenzar la búsqueda de la ineficiencia más importante en la ejecución de la aplicación. Entendemos como ineficiencia aquel intervalo de tiempo con menor número de procesadores activos durante mayor número de tiempo. Esta búsqueda tendrá como resultado una lista de intervalos de ejecución que contienen potenciales problemas de rendimiento y que deberán ser analizados en busca de sus causas.

3. Para cada una de las ineficiencias obtenidas se construye una identificación de los procesos involucrados y se recoge el tiempo de bloqueo que esta situación genera a lo largo de toda la ejecución. Si todo este tiempo recogido no supone un porcentaje mínimo (configurable) de la ejecución no se considera lo suficientemente importante para buscar sus causas en las siguiente etapas. Por lo tanto, solo se van a considerar como importantes aquellos problemas que están presentes durante una parte significativa de la ejecución.
4. Las ineficiencias que más influyen en la ejecución son analizadas una por una en busca de sus causas. El fichero de traza es de nuevo estudiado recogiendo todos los eventos de ejecución en los intervalos de ejecución donde se produce la ineficiencia. Se identifican los procesos que provocan el problema y se analizan en detalle las acciones que los procesos realizan en ese momento. Finalmente, se utiliza la "base de conocimiento" de los problemas de rendimiento para encontrar las causas de los problemas encontrados.

2.3. Sugerencias al usuario.

Cuando el análisis ha obtenido una serie de causas para los problemas más importantes de rendimiento de la aplicación, se empiezan a construir una serie de sugerencias para que el usuario entienda dónde se encuentran los problemas más importantes de la ejecución. Estos informes incluyen información respecto a dónde se ha encontrado este problema del rendimiento en la ejecución; cuáles son las posibles causas de problemas del rendimiento y cómo se relaciona el problema encontrado con el código de la aplicación (procesos relacionados con el problema, referencias al código fuente del programa como estructuras de datos usadas, líneas de código, etc…).
Junto con estas explicaciones, la herramienta propone una serie de recomendaciones al usuario que dependerán mucho del tipo de problema. Estas recomendaciones pretenden enfocar al usuario en una posible modificación del código que mejore el rendimiento de la aplicación.

3. Clasificación de los problemas

El principal aspecto del análisis automático es su uso de una lista predefinida de problemas de rendimiento. En esta sección vamos a introducir los problemas de rendimiento encontrados estudiando el modelo de programación PVM y sus aplicaciones. Si la herramienta resulta útil para los programadores de C más PVM, podemos acercarnos a otras librerías y otros modelos con el mismo propósito.

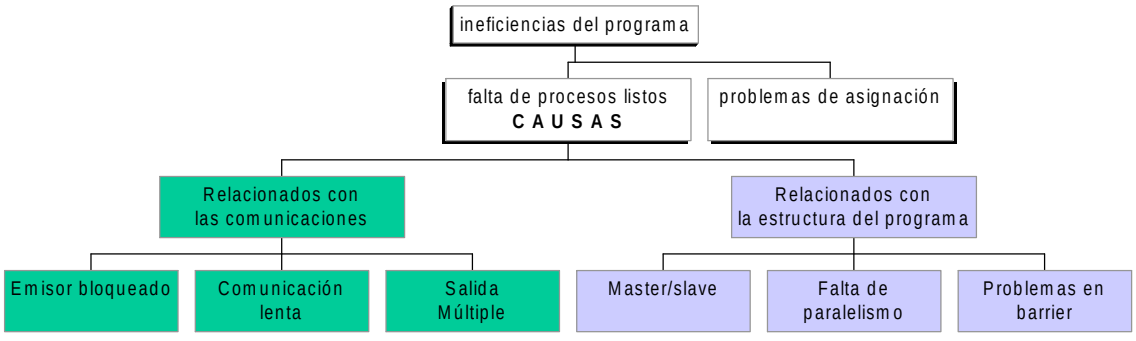

Figura 3: Clasificación de los problemas de rendimiento de los programas paralelos.

El análisis de los patrones de ineficiencia hace una diferenciación inicial entre dos tipos de problemas básicos:

- Falta de procesos listos. Situaciones en las que hay procesadores libres y no hay procesos listos para ejecutar ni procesos que compartan un mismo procesador del sistema. La idea para saber qué impide a la aplicación ocupar todos los procesadores es encontrar qué

cambios pueden realizarse en la aplicación para activar más procesos en esos intervalos de inactividad.

- Problema de asignación. Mientras se produce una situación de ineficiencia existen algunos procesadores desocupados. En este tipo de situaciones, además existen procesos listos para ejecutar asignados a la cola de un procesador que ya está ocupado. Este problema de balanceo de carga se incluye en esta clasificación pese a que no suele producirse en programas PVM (donde se puede usar un mecanismo automático de balanceo de procesos).

Cada situación de ineficiencia será clasificada en una de estas dos clases. De cualquier forma, estas dos situaciones no son mutuamente exclusivas. Es posible que la herramienta encuentre un intervalo con un rendimiento pobre con procesos listos asignados a procesadores ocupados. Cuando la herramienta intenta determinar qué proceso debe ser asignado a algún procesador libre podría encontrarse con que no es posible reasignar los procesos a otros procesadores debido a sus dependencias de comunicación. Si alejamos un proceso de aquellos con los que se comunica, hacemos que estos mensajes necesiten más tiempo para llegar a su nuevo destino. Si este es el caso y la mejoría de rendimiento se prevee mínima, la herramienta volverá a tratar el problema como una falta de procesos listos.

La clasificación de problemas de rendimiento y sus causas pueden verse en la figura 3. En los siguientes puntos estos problemas serán descritos de acuerdo con la clasificación de la figura 3.

3.1. Falta de procesos listos

Si no hay procesos listos para ejecutarse en los procesadores libres del sistema debemos buscar las razones por las que los procesos que se ejecutan justo después de la ineficiencia no se ejecutan antes y qué debería cambiarse en el diseño de la aplicación para activarlos antes (llenando así el intervalo de inactividad).

La experimentación realizada muestra que las causas de un problema de falta de procesos listos puede dividirse en dos categorías (siguiendo con el siguiente nivel en la figura 3): los problemas relacionados con las comunicaciones y los relacionados con la estructura del programa. De esta forma, vamos a analizar estas ramas del árbol de causas con mayor detalle, describiendo la manera de funcionar de la herramienta para cada problema.

3.1.1. Emisor Bloqueado

Esta situación de ineficiencia viene definida por una serie de bloqueos encadenados por comunicaciones. Cuando, al analizar la historia de una transferencia, descubrimos que el emisor del mensaje (por el cual se está esperando) está a su vez esperando por la llegada de un mensaje de un tercer proceso. Este encadenamiento de bloqueos está compuesto de, como mínimo, tres procesos. Por lo tanto, la herramienta deberá concentrarse en el análisis de dependencias entre los procesos relacionados por estos bloqueos, en busca de alguna relación que pueda cambiar esta situación de ineficiencia.

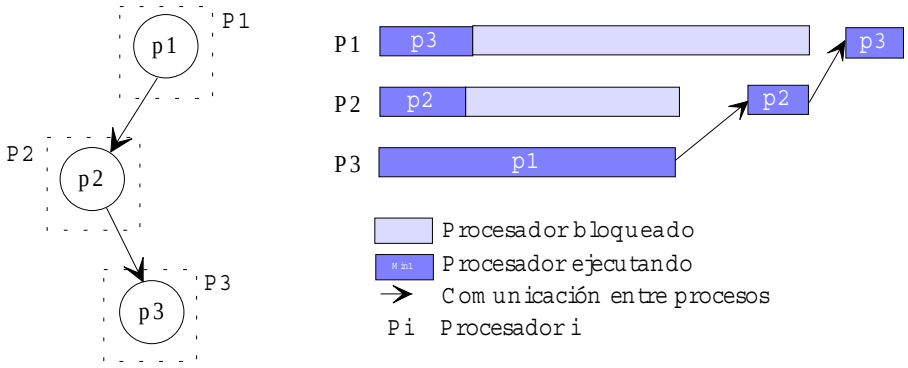

Figura 4: Ejemplo de un problema de emisor bloqueado

En el ejemplo de la figura 4, la herramienta detecta una relación de bloqueo de comunicación entre tres procesos. Se encontrará una dependencia de datos cuando el proceso p3 se bloquea por la recepción de un mensaje que el proceso p2 debe enviar. Al mismo tiempo, el proceso p2 se bloquea para recibir un mensaje del proceso p1. Si estos dos mensajes se relacionan de alguna forma (por ejemplo, si un mensaje forma parte del otro) la herramienta notificará esta situación y sugerirá cómo distribuir los datos de una forma que ofrezca posibilidades de un mejor rendimiento en la ejecución. En la sección 4, se profundizará en la detección y análisis de un problema de este tipo.

3.1.2 Comunicaciones Lentas.

Un intervalo de inactividad también puede ser causado por una comunicación entre dos procesos que se enlentece tanto que llega a serializar la ejecución de la aplicación. Si esta serialización se produce repetidas veces durante la ejecución, se convertirá en una ineficiencia importante para la herramienta. Una vez reconocida como problema, esta situación se analiza reconstruyendo la historia de las comunicaciones entre los procesos que se comunican. Este problema de comunicación lenta será considerado cuando al detectar que el tiempo de ineficiencia coincide con el tiempo en el que el mensaje ha estado moviéndose a través de la red de interconexión.
Las soluciones a este problema pasan por considerar posibles cambios en la asignación de los procesos a procesadores más cercanos, considerando los niveles de carga de los procesadores cercanos, para intentar influir lo mínimo en el balance de carga del sistema.

3.1.3 Problema de salida múltiple

Otro de los problemas de comunicación se produce cuando un proceso debe enviar una serie de mensajes a un grupo de procesos (operaciones de broadcast, multicast o scatter por ejemplo).Si, tal como muestra la figura 5, una serie de procesos se bloquean (al mismo tiempo) para recibir un mensaje, podemos estar delante de una serialización de las comunicaciones, que puede llegar a afectar negativamente al rendimiento de la aplicación.
La figura 5 representa una construcción típica de salida múltiple. Este problema puede surgir al utilizar las primitivas de “send” y “recv” para construir comunicaciones globales. Cuando el proceso “bill” implementa la operación “multicast” de una forma más eficiente (enviando los mensajes a la vez) esta estructura no generará un problema relevante. Si se utilizan las primitivas de comunicación punto a punto y todos los procesos que esperan se bloquean a la vez irán recibiendo los mensajes uno a uno. Estos mensajes desbloquearán a los receptores de forma secuencial hasta que llegue el último mensaje. Ese último proceso será el que más espere de todos los receptores.
Cada vez que la herramienta analice una ineficiencia y detecte un bloqueo de espera de mensaje mientras que el proceso que debe enviarlo está enviando mensajes a otros procesos en la aplicación, tendremos un problema de salida múltiple.

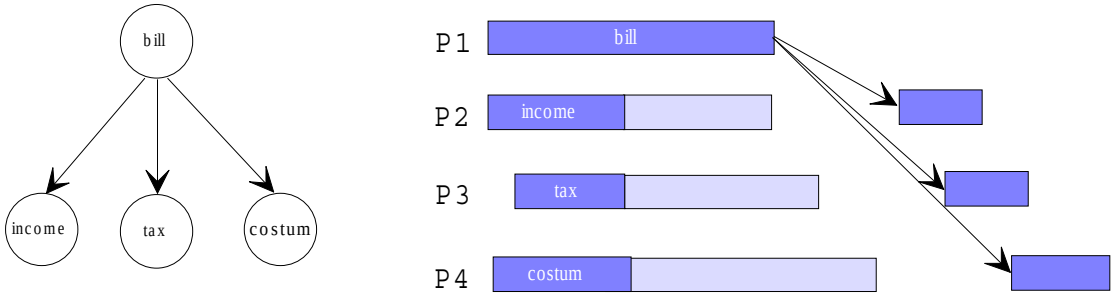

Figura 5: Esquema de salida múltiple cuando se consideran primitivas bloqueantes de recepción de mensaje.

Cuando la herramienta se encuentra con un problema de salida múltiple, sugerirá la posibilidad de utilizar una primitiva de "multicast" para los mensajes. En cambio, si la herramienta encuentra diferencias entre los datos enviados, sugerirá el uso de una operación de "scatter" para distribuir los mensajes disjuntos.

3.1.4 Colaboraciones master/slave problemáticas

Una característica especial de la herramienta es la posibilidad de buscar colaboraciones master/slave en la ejecución de la aplicación. En estos esquemas, un proceso master genera un dato y lo envía a un proceso esclavo para que sea procesado. Inmediatamente después de que el master envíe el dato, comienza a generar un nuevo dato para enviarlo.

A partir de estas reglas de funcionamiento, si el master genera datos frecuentemente, el número de esclavos debería ser suficientemente alto como para cubrir esa necesidad de procesamiento. De aquí, la herramienta intenta analizar cuántos esclavos necesita el master, sugiriendo un número de esclavos que parezca adecuado para mejorar el rendimiento.
Para detectar las necesidades del master, la herramienta localiza los procesos que colaboran en el cálculo y analiza los detalles siguientes:
- Tiempos de bloqueo de los procesos master. Si es muy alto, la herramienta recomendará la creación de nuevos esclavos. Cuanto menos tiempo esperen los procesos master, mejor rendimiento tendrá la aplicación.
- Los tiempos de espera por mensaje (cuando éste ya está transmitiéndose). Si la comunicación es demasiado lenta, (los procesos esclavos esperan por una importante cantidad de tiempo mientras los mensajes viajan a través de la red de interconexión), la herramienta puede recomendar revisar la política de asignación de forma que los procesos esclavos estén lo más cerca posible de los master. Por otro lado, también se sugiere un límite en el número máximo de esclavos para evitar la contención en una red de interconexión saturada de mensajes.

3.1.5 Falta de paralelismo.

Un problema de falta de procesos listos para ejecutar se presenta como una situación en la que no hay suficientes procesos disponibles como para llenar todos los procesadores del sistema. Por lo tanto, es una situación que depende de la estructura del programa que se está ejecutando. En la mayoría de estas situaciones, el problema ha sido causado por un diseño que no genera un número suficiente de procesos. Por lo tanto, la recomendación principal siempre pasa por rediseñar el programa distribuyendo el cálculo a realizar de una forma que serialice menos la ejecución. De esta forma, la herramienta KAPPA-PI, una vez detecta esta situación, focaliza sus esfuerzos en encontrar una razón por la que los procesos que siguen a la inactividad no pueden ejecutarse antes.
Normalmente, tal y como se muestra en la figura 6, estos procesos dependen de otros para ejecutarse, debiendo esperar a que sea iniciada su ejecución. Por ejemplo, mediante el uso de la

primitiva “spawn”. La herramienta KAPPA-PI se encargará de buscar qué acciones realizan los procesos que deben llamar a esa primitiva y que demoran la creación del nuevo proceso. Si se encuentra algún evento especial se sugerirá el rediseño de la aplicación. En cambio, si no es así se considerará como un problema de serialización.

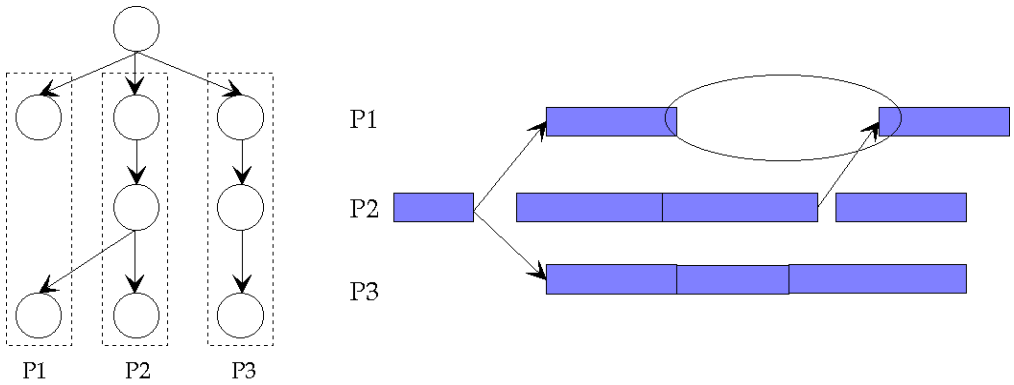

Figura 6: Falta de paralelismo en el segundo nivel del diseño.

3.1.6 Problemas con las primitivas “barrier”

El uso habitual de las primitivas de barrera o “barrier” se basa en la llamada a la primitiva desde uno de los componentes de un grupo de procesos que colaboran en una tarea específica. La primitiva barrier bloquea a todos los procesos que van haciendo la llamada hasta que un cierto número predefinido de procesos ya haya llamado a la barrera. Cuando un proceso llega a la barrera considerablemente más tarde que el resto de los procesos, éstos deberán esperarla. Bloqueando así la ejecución de la aplicación.

La herramienta, al analizar un problema de este tipo, focaliza su trabajo en encontrar una causa a la última llegada a la barrera. Primero descubre qué tarea llega la última a la barrera. Dedicándose a recoger todos los eventos de ejecución de los últimos procesos que llegaron a la barrera. De estos eventos, la herramienta obtiene las razones por las que el proceso llega tarde a la barrera. Estas razones, (ya sea una comunicación o un cálculo excesivamente largo) son presentadas al usuario como las razones que provocan ese bloqueo reiterado en la barrera, siendo los puntos que deberían modificarse en las siguientes versiones del diseño.

3.2 Problema de Asignación.

Cuando, al analizar los patrones de ineficiencia de la ejecución, la herramienta encuentra algún proceso listo para ejecutar en un procesador que ya está ocupado, trata de reasignar los procesos intentando llenar los procesadores vacíos del sistema.

Primero, la herramienta considera todos los procesos susceptibles de ser reasignados a los procesadores vacíos. Después, se generan todas las parejas “proceso - nuevo procesador“ para formar una lista de parejas candidatas a aplicar en una nueva asignación. En una tercera etapa, cada pareja formada se analiza para valorar el posible incremento de rendimiento al aplicar el cambio en la asignación.
El propósito de la herramienta es redistribuir los procesos listos para ejecutar en los procesadores libres del sistema. Para cada cambio en la asignación, la herramienta generará una sugerencia al programador de la aplicación.
Hay que tener en cuenta que, en la ejecución de los programas PVM, estas sugerencias sobre la asignación de los procesos sólo se aplica en aquellos programas en los que se está haciendo la asignación de forma manual, no utilizando la utilidad de balanceo automático de carga proporcionado por la función “spawn”. De cualquier forma, las recomendaciones generadas por la herramienta deberían ser usadas como entrada a un asignador de procesos incluido en el sistema paralelo y no sería la clase de información que debería ser presentada al usuario.

4. Ejemplo.

En la figura 7 presentamos un ejemplo de análisis de rendimiento de una aplicación. La herramienta comienza el análisis buscando intervalos de ineficiencia y encontrando la más importante en el intervalo (t1, t2). Este intervalo está formado por dos bloqueos por comunicaciones que se repiten a lo largo de la ejecución. Por simplicidad, no vamos a mostrar toda la ejecución completa del programa, centrándonos en el análisis concreto de la situación encontrada.
Una vez la herramienta ha determinado la influencia del problema de rendimiento sobre la ejecución, la siguiente etapa es recoger los detalles de ejecución de la aplicación para averiguar las causas de la ineficiencia encontrada.
Dado que en los procesadores P2 y P3 no hay procesos listos para ejecutarse en el intervalo de tiempo (t1, t2) la herramienta trata de encontrar la causa de esa falta de procesos listos para ejecutar. Cuando la herramienta detecta que la ineficiencia incluye una comunicación (es decir, que los bloqueos son esperas de mensaje), construye una tabla donde almacena las comunicaciones que se establecen durante los intervalos de ineficiencia a partir de los eventos de comunicaciones del fichero de traza.
En el ejemplo, la herramienta detecta que el proceso “Max0” está esperando por un mensaje desde el proceso “Min1”, el cual también está esperando a su vez por otro mensaje que debe llegar desde el proceso “Max2”. Esta situación cumple con los supuestos de la situación de “Emisor Bloqueado”. De esta forma, la herramienta asume que ha encontrado un problema de Emisor Bloqueado.

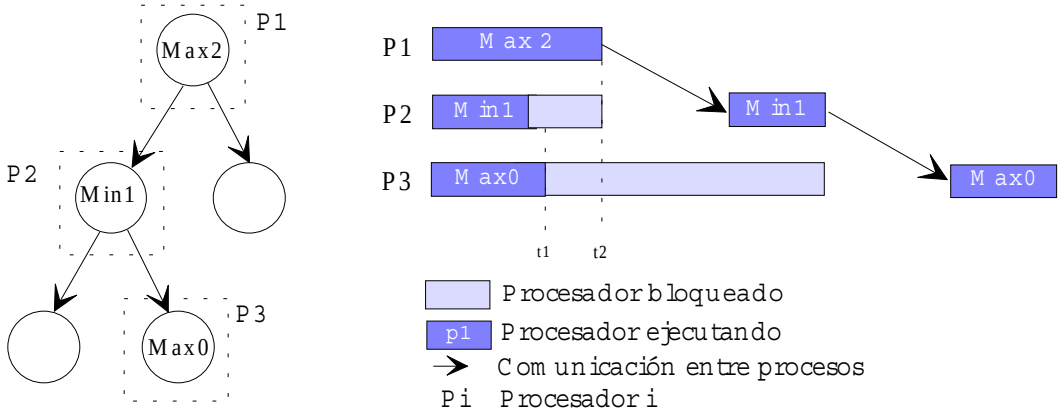

Figura 7: aplicación y representación del fichero de traza de su ejecución.

A) Los dos procesos necesitan el mismo dato.

Una vez la herramienta ha detectado el problema de rendimiento y su causa, procede a analizar los detalles de ese problema encontrado, intentando localizar todas las posibles relaciones de la ejecución con el propio código de la aplicación. En este caso, la herramienta accede al código de los tres procesos relacionados por las comunicaciones y analiza la naturaleza de los mensajes enviados en ese tiempo de bloqueo. En concreto, accede al código del proceso “Min1” , en busca de las emisiones y recepciones de los mensajes problemáticos. En la figura 8 se puede comprobar cómo la herramienta encuentra que la aplicación está enviando el mismo mensaje desde “Max2” a “Min1” y de “Min1” a “Max0”. Por lo tanto, una solución a este problema de rendimiento pasa por rediseñar estas comunicaciones encadenadas.

```
1  pvm_recv(-1,-1);
2  pvm_upkfl(&calc,1,1);
3  calc1 = calc;
4  for(i=0;i<sons;i++)
5  {
6     pvm_initsend(PvmDataDefault);
7     pvm_pkfl(&calc1,1,1);
8     pvm_send(tid_son[i],1);
9  }
```

Figura 8: Análisis del código del proceso “Min1”.

En el ejemplo, el programador recibe un mensaje de la herramienta donde se explica el problema de rendimiento encontrado y se sugiere una posible actuación por su parte para corregir la deficiencia. En este caso, la sugerencia mostrará una posible modificación del código para reestructurar las comunicaciones. En la figura 9 podemos comprobar la recomendación proporcionada. Como la herramienta encontró que se estaba enviando el mismo mensaje a los proceso “Max0” y “Min1”, se sugiere rediseñar las comunicaciones entre esos procesos mediante el uso de una primitiva “multicast” para enviar esos dos mensajes (que provocan un bloqueo encadenado) en paralelo.

```
Se ha detectado una recepción en “Min1” mientras “Max0” espera por un mensaje de “Min1”.
Se ha detectado un problema de dependencias al analizar un bloqueo en “Max0”.

Los procesos implicados en la dependencia, que deberían ser revisados, son
> Proceso “Min1”, que espera por un mensaje de “Max2” en linea 25.
> Proceso “Max0”, que espera por un mensaje de “Min1” en linea 39.

Se ha detectado que los mensajes transmitidos son el mismo. Se recomienda utilizar una
    primitiva “multicast” en el proceso “Max2”.
```

Figura 9: Recomendación proporcionada al usuario por la herramienta KAPPA-PI.

En la figura 10 podemos comprobar los resultados de modificar la aplicación previamente analizada. La herramienta sugiere el uso de una primitiva “multicast” cuando el mensaje enviado por “Min1” es exactamente el mismo que el recibido en este mismo proceso. Para mejorar el rendimiento, la herramienta sugiere establecer un enlace entre los procesos “Max2” y “Max0” para enviar el mensaje que está siendo reenviado por el proceso “Min1”. Al obtener los nuevos datos de ejecución se detecta un nuevo problema de “Salida Múltiple”, esta vez con un menor impacto en el rendimiento de la ejecución.

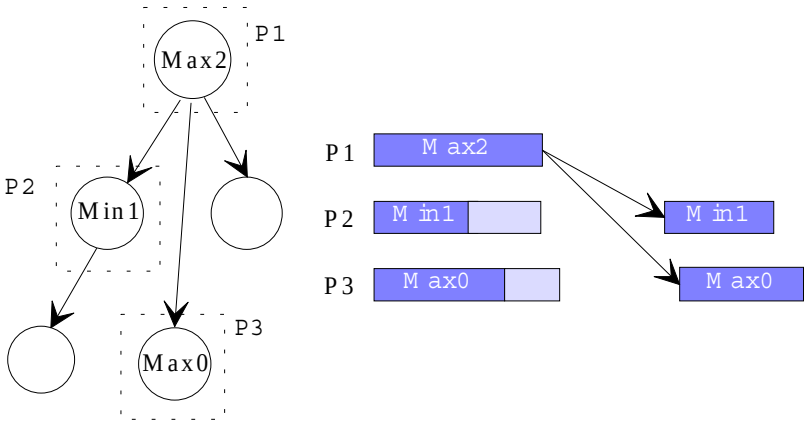

Figura 10: Ejecución de la aplicación al aplicarle el cambio propuesto por la herramienta.

Si la herramienta encuentra otras configuraciones en el esquema de envíos y recepciones del código de la aplicación proporcionará diferentes sugerencias para rediseñar el comportamiento de la aplicación. En concreto, son las dos situaciones siguientes:

B) No hay dependencias entre los procesos que se bloquean por la comunicación

En este caso, la herramienta detecta que “Max0” espera bloqueado innecesariamente hasta que la comunicación entre “Max2” y “Min1” ha terminado. Si el programador intercambiara las primitivas de recibir y enviar en el proceso “Min1”, el problema de emisor bloqueado desaparecería (figura 11). Por lo tanto, la herramienta sugiere esta alternativa cuando el mensaje enviado a “Min1” es considerado como independiente del mensaje previamente recibido (no hay asignaciones entre los dos datos en el código que hay entre los envíos).

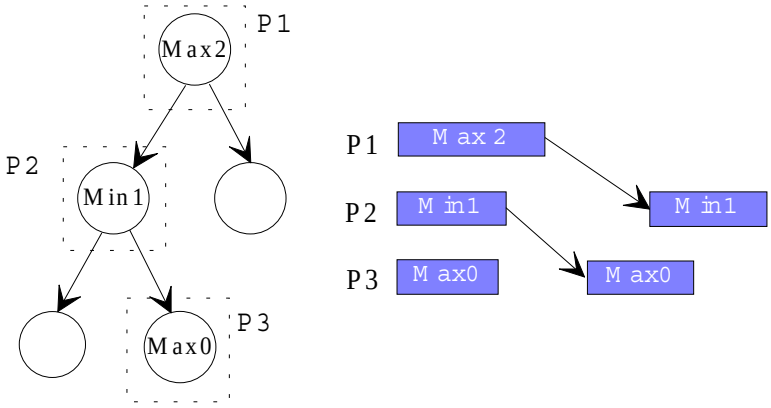

Figura 11: Ejecución de la aplicación cuando no hay dependencias entre los procesos.

C) Existen dependencias de datos entre los procesos.

En este caso, la herramienta encuentra que el mensaje recibido en “Min1” es modificado y luego enviado al proceso “Max0”.
Por lo tanto, el proceso “Max0” debe esperar a que “Min1” reciba su mensaje, le aplique la modificación y lo envíe. Tomando esta modificación del dato como inamovible en el diseño de la aplicación, la herramienta sugiere enviar el mensaje a “Min1” lo antes posible, intentando así reducir el tiempo de espera por el mensaje “Max0”. En la figura 12 podemos comprobar cuál sería el resultado de aplicar estas modificaciones a la aplicación.

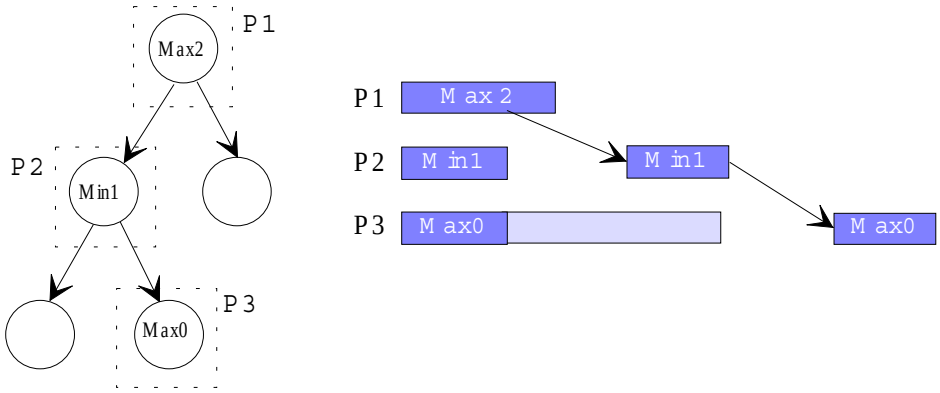

Figura 12: Ejecución de la aplicación cuando existe una dependencia de datos.

5.- Conclusiones y trabajo futuro.

Esta herramienta automática de análisis de rendimiento está diseñada para programadores de aplicaciones paralelas que quieren mejorar el comportamiento de sus aplicaciones. La visión de los programadores de la herramienta es bastante simple: la ejecución de la aplicación es llevada al análisis como entrada, obteniéndose una serie de sugerencias como mejoras a realizar en el código de la aplicación para así obtener un mejor rendimiento del programa. Estas sugerencias explican, al nivel del programador, qué problemas de rendimiento han sido encontrados en la ejecución y cómo resolverlos modificando el diseño de la aplicación.

De cualquier forma, al aplicar los cambios propuestos al código de la aplicación pueden surgir otros nuevos problemas de rendimiento. Los programadores deben conocer los posibles efectos laterales de introducir cambios en las aplicaciones. De aquí, una vez se reconstruye el diseño de la aplicación, se debe considerar una nueva etapa de análisis. Este nuevo análisis debe ser testeado para encontrar un nuevo conjunto de datos de entrada para así analizar la ejecución de la aplicación de una forma completa a partir de un fichero de traza.

El futuro trabajo en la herramienta pasa por completar la base de conocimiento de problemas del rendimiento. En esta línea, existe una necesidad de una metodología que permita establecer cuándo una herramienta de análisis de rendimiento es útil para el programador. Es decir, que la herramienta detecta una serie de problemas que se consideran importantes y además sirve para mejorar el diseño de la aplicación. En estos momentos este proceso de evaluación de una herramienta de este tipo no está bien definido, aunque existen ya una serie de programas-prueba, como “Grindstone” definidos en [Hol 96], cuya detección está siendo incluida dentro de la base de conocimiento.

Desde luego, también es necesario enfocar el trabajo de análisis de los programas a otras clases de librerías de paso de mensajes con diferentes problemáticas, caso de MPI.

Finalmente, también es necesario considerar la dificultad de analizar aplicaciones durante un cierto tiempo de ejecución debido a la dificultad de manejar grandes ficheros de traza. Es necesario replantear la instrumentación de forma que este problema se minimice dentro de lo posible, eliminando la información redundante que no se va a utilizar en el análisis.

6- Referencias

[Cro 94]: Crovella, M. E. and T. J. LeBlanc. “*The search for Lost Cycles: A New approach to parallel performance evaluation*”. TR479. The University of Rochester, Computer Science Department, Rochester, New York, December 1994.

[Fah 96]: Fahringer, T. 1996 *Automatic Performance Prediction of Parallel Programs*. Kluwer Academic Publishers.

[Hol 93]: Hollingsworth, J. K. and B. P. Miller. “*Dynamic Control of Performance Monitoring on Large Scale Parallel Systems*”. International Conference on Supercomputing , Tokyo, July 1993.

[Hol 96]: Hollingsworth, J. K. and Steele, M. “Grindstone: A Test Suite for Parallel Performance Tools”. Technical Report CS-TR-3703, University of Maryland, 1996.

[Hea 95]: Heath, M T., J. A. Etheridge: "*Visualizing the performance of parallel programs*". IEEE Computer, November 1995, vol. 28, p. 21-28 .

[Mai 95]: Maillet, E: ”*TAPE/PVM an efficient performance monitor for PVM applications-user guide*”, LMC-IMAG Grenoble, France. June 1995.

[Mei 95]: Meira, W Jr. “*Modelling performance of parallel programs*”. TR859. Computer Science Department, University of Rochester, June 1995.

[Pan 95]: Pancake, C. M., M. L. Simmons, J. C. Yan: "*Performance Evaluation Tools for Parallel and Distributed Systems*". IEEE Computer, November 1995, vol. 28, p. 16-19.

[Ree 93]: Reed, D. A., 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*". Proceedings of Scalable Parallel Libraries Conference. IEEE Computer Society, 1993.

[Wal 96]: Wall, L., T. Christiansen, R. L. Schwartz: *Programming Perl*. O'Reilly and Associates, 2nd Edition, Nov 96.