Gestión dinámica de aplicaciones master-worker sobre sistemas distribuidos

E. Heymann, M. A. Senar y E. Luque
Unitat d'Arquitectura d'Ordinadors i Sistemes Operatius
Universitat Autònoma de Barcelona
Barcelona, Spain
{e.heymann, m.a.senar, e.luque}@cc.uab.es

Resumen

Las aplicaciones paralelas que obedecen al paradigma master-worker necesitan de políticas de gestión con objeto de maximizar el rendimiento de la aplicación a la vez que se hace un uso eficiente de los recursos de cómputo. En este trabajo proponemos una política simple que es capaz de realizar la gestión dinámicamente de aplicaciones master-worker a partir de la información que se obtiene en tiempo real sobre la misma. La política es capaz de ajustar el número de procesadores utilizados por la aplicación con objeto de mejorar la eficiencia en el uso de recursos sin penalizar excesivamente el tiempo de ejecución final. La bondad de la política ha sido comprobada mediante un extenso conjunto de simulaciones donde se consideraron múltiples factores para modelar el comportamiento de las aplicaciones master-worker. Describimos también los resultados obtenidos por un primer prototipo de la política ejecutado sobre un conjunto homogéneo de estaciones de trabajo que usaban el sistema Condor como gestor de recursos.

Palabras clave: Gestión de tareas, balanceo dinámico de carga, aplicaciones *master-worker*, computación distribuida, *cluster computing*, *metacomputing*.

1. Introducción

El uso de computadores potentes y de bajo coste, como PCs o estaciones de trabajo, interconectados por redes de alta velocidad está derivando en los últimos años en el uso masivo de estas plataformas para resolver problemas que requieren mucha potencia de cómputo. Bajo la denominación genérica de *cluster computing* [Buy99] se engloban diferentes líneas de investigación que estudian la forma de utilizar las agrupaciones (*clusters*) de máquinas de forma eficiente. Estos sistemas se han utilizado para desarrollar aplicaciones paralelas aunque, en general, no todos los programas paralelos que se ejecutan eficientemente en un supercomputador paralelo tradicional pueden ser trasladado a un *cluster* sin que sufran pérdidas significativas de rendimiento. El paradigma master-worker es uno de los más atractivos para su ejecución en clusters porque puede ser adaptado fácilmente de forma que se consiga un rendimiento similar al de los supercomputadores tradicionales [SB99].

En este paradigma, un proceso *master* es responsable de distribuir un conjunto de tareas entre un conjunto de procesos trabajadores (*workers*). Normalmente, las aplicaciones master-worker no exhiben muchas comunicaciones lo que las hace apropiadas para la mayoría de clusters ya que no se requiere un rendimiento elevado de la red de interconexión. Además, este paradigma es un buen ejemplo de cómputo paralelo adaptable ya que puede responder a situaciones donde las aplicaciones se ejecutan aprovechando ciclos de procesadores desocupados (en entornos conocidos como clusters no dedicados oportunísticos). El número de procesos trabajadores puede adaptarse en tiempo de ejecución al número de recursos presentes en un entorno oportunístico de forma que si aparecen nuevos recursos, éstos sean utilizados como nuevos trabajadores para la aplicación.

Sin embargo, el uso de un cluster no dedicado introduce la necesidad de utilizar mecanismos complejos que permitan descubrir recursos disponibles, reservarlos, migrar procesos y equilibrar la carga global del sistema. Una aplicación paralela necesita algún medio para descubrir qué recursos

están desocupados y necesita también algún mecanismo para asignarse el recurso y crear un worker allí. Cuando el propietario del recurso lo vuelve a reclamar, la tarea que se estuviese ejecutando en él debe abandonar el procesador y, eventualmente, debe reasignarse a otro worker cuando éste finalice su tarea actual. Normalmente, estos servicios son ofrecidos por una capa intermedia de software situada entre la aplicación del usuario y el sistema operativo denominada comúnmente *cluster middleware*. Lógicamente, los servicios proporcionados por el *cluster middleware* suponen un coste adicional que debe añadirse al tiempo de ejecución de la aplicación.

En el caso de las aplicaciones master-worker, este coste adicional introducido por el *middleware* pueden compensarse haciendo que los recursos asignados se mantengan durante todo el tiempo de ejecución de la aplicación. Esta política de gestión de recursos deriva en una estrategia de tipo ávido donde la aplicación intenta conseguir tantos recursos como sean posibles para utilizarlos posteriormente. Esto provoca situaciones donde a lo largo de la ejecución de la aplicación diversos workers están inactivos mientras que otros workers todavía deben finalizar sus tareas. Esto conlleva una utilización pobre de los recursos porque no todos se mantienen ocupados haciendo trabajo útil durante la mayor parte del tiempo y, por lo tanto, la eficiencia de la aplicación es baja. La forma de mejorar la eficiencia en estos casos es restringir el número de workers asignados.

Si tenemos en cuenta el tiempo de ejecución de la aplicación, la asignación de workers suele guiarse por criterios diferentes ya que la aceleración (*speedup*) que experimenta la aplicación depende directamente del número de workers que se utilicen: cuantos más workers se usan menor será el tiempo de ejecución total de la aplicación. En general, la ejecución de una aplicación master-worker debe mostrar un compromiso entre el speedup conseguido y la eficiencia exhibida. La aplicación debe ejecutarse tan rápido como sea posible a la vez que todos los workers se mantienen ocupados con trabajo útil.

En este trabajo nuestro objetivo se centra en maximizar la eficiencia de una aplicación master-worker, tratando de producir el menor impacto posible en la pérdida de speedup de la aplicación. Esto se conseguirá, por una parte, controlando el número de workers utilizados por la aplicación y, por otra parte, gestionando la asignación de las tareas a los workers. Este objetivo lo abordamos, en primera instancia, proponiendo un modelo de aplicaciones master-worker que nos servirá posteriormente para definir la política de gestión que permita controlar de forma dinámica el número de workers y la asignación de las tareas de la aplicación. Hemos evaluado la bondad de la política de gestión tanto por simulación como mediante un prototipo real ejecutándose sobre un cluster de máquinas Linux.

El resto del artículo está organizado de la siguiente forma: en la sección 2 se describe las características del modelo master-worker que consideramos en este trabajo; la sección 3 presenta de forma más precisa el problema de la gestión de las tareas de una aplicación master-worker, introduce nuestra política y revisa brevemente otros trabajos relacionados; la sección 4 presenta la evaluación llevada a cabo mediante simulación de la política propuesta; la sección 5 describe los resultados obtenidos con un prototipo operativo de la política; por último, la sección 6 resume las aportaciones de este trabajo y describe las líneas de trabajo que se van a seguir.

2. El modelo de las aplicaciones master-worker

En este trabajo, nos centramos en el estudio de aplicaciones master-worker con un modelo similar de comportamiento y que ha sido utilizado para resolver un número significativo de problemas tales como simulaciones del problema de los N-cuerpos [GF96], simulaciones por el método de Monte Carlo [BRL99] y simulaciones de materiales [PL96]. En la figura 1.1 mostramos el esquema algorítmico correspondiente a este modelo, en el cual el proceso master resuelve iterativamente un lote de tareas. Para ello buscará procesos worker que las puedan ejecutar. A un proceso worker el master le enviará los datos de entrada de la tarea y esperará recibir de él los resultados finales de la

ejecución de la tarea. Cuando una tarea finaliza, el proceso master puede realizar algún cómputo intermedio con el resultado parcial generado por la tarea. Posteriormente, cuando todo el lote de tareas finalice, el master puede llevar a cabo algún procesamiento adicional. Después de eso, el master pasará a resolver un nuevo lote de tareas, repitiéndose este proceso varias veces hasta que se alcance la resolución del problema. En general, denominaremos K al número de iteraciones en las que el proceso master repite el proceso descrito anteriormente.

```
Inicialización
  Do
        For tarea = 1 to N
            ResultadoParcial  =  + Funcion (tarea)
        End
        Actuar_ante_lote_completo( )
  while (condición final no alcanzada).
```

Figura 1.1. Modelo de aplicaciones master-worker.

En este tipo de aplicaciones un cierto lote de tareas paralelas se ejecuta K veces (iteraciones). La finalización de un cierto lote introduce un punto de sincronización en la aplicación que facilita la recolección de información estadística por parte del proceso master del funcionamiento de los procesos workers.

Diferentes autores han constatado empíricamente que muchas aplicaciones master-worker que siguen el modelo expuesto anteriormente acostumbran a presentar un comportamiento regular en el que la ejecución de una misma tarea a lo largo de las distintas iteraciones suele tener un comportamiento similar; lo que se traduce en tiempos de ejecución parecidos en las distintas iteraciones [PL96]. Esto supone que este tipo de aplicaciones muestra un elevado grado de predictibilidad, de forma que las medidas tomadas en una cierta iteración pueden proporcionar buenas predicciones sobre el comportamiento futuro. En las siguientes secciones de este trabajo mostraremos hasta qué punto una estrategia adaptativa de gestión de tareas puede hacer uso de la información histórica sobre el comportamiento de una aplicación master-worker con objeto de mejorar su rendimiento en un entorno distribuido.

2. El problema de la gestión de aplicaciones Master-Worker

En esta sección vamos a formalizar el problema de la gestión de aplicaciones master-worker, enmarcándolo en los trabajos relacionados con este tema y presentando la política de gestión que se propone para este tipo de problema.

3. 1 Planteamiento del problema y trabajos relacionados

La ejecución eficiente de una aplicación master-worker como la presentada en el apartado anterior debe responder, en primera instancia, a dos cuestiones fundamentales: ¿cuántos procesos trabajadores va a utilizar la aplicación? y ¿cómo van a asignarse las tareas a los trabajadores?

La primera pregunta admite una respuesta muy simple que consiste en utilizar tantos procesos trabajadores como tareas contenidas en un lote (N). Como ya se comentó anteriormente, esta estrategia va a producir un uso pobre de los recursos porque, si las tareas presentan tiempos de ejecución diferentes, aquellos workers a los que se les asignen tareas cortas acabarán antes y permanecerán inactivos esperando la finalización de los workers a los que se les a asignado tareas largas. En consecuencia, una política que busque un uso eficiente de los recursos tenderá, en general, a limitar el número de workers utilizados, de forma que la aplicación se ejecute sólo sobre W workers (siendo $W < N$).

De acuerdo con la decisión tomada para responder a la primera pregunta (usar W workers para ejecutar las N tareas), la respuesta que se dé a la segunda pregunta va a condicionar el tiempo final de ejecución de cada uno de los lotes de tareas. Si el número de workers es menor al número de tareas, algún worker va a resolver más de una tarea y en esta situación es donde el orden de asignación de tareas va a ser importante. Un orden erróneo, por ejemplo, puede provocar que muchos workers queden inactivos mientras que a unos pocos se les asigna al final las tareas más largas del lote.

La resolución de las dos preguntas formuladas anteriormente es muy compleja. Baste mencionar que la segunda pregunta es equivalente a la formulación del problema de asignación de tareas independientes en máquinas idénticas con objeto de minimizar el tiempo de finalización total (problema conocido como *off-line scheduling*). Este problema es de tipo NP-completo, por lo que no se conocen algoritmos que lo resuelvan de forma óptima y sólo se dispone de algoritmos que proporcionan soluciones aproximadas [Hal99]. En nuestro caso, la complejidad del problema es mayor porque existe un cierto grado de indeterminación al no conocerse a priori el tiempo exacto que va a tardar en ejecutarse una determinada tarea.

La evaluación de una cierta política de gestión se basará en dos parámetros: la eficiencia y el tiempo de ejecución de la aplicación master-worker. La eficiencia (E) en el uso de recursos para W workers se puede definir como la proporción entre el tiempo en el que cada worker ha estado realizando trabajo útil y el tiempo en el que los workers estuvieron disponibles para hacer trabajo (ec. 1):

$$E = \frac{\sum_{i=1}^{W} T_{work,i}}{\sum_{i=1}^{W} T_{up,i} - \sum_{i=1}^{W} T_{susp,i}} \qquad \text{(ec. 1)}$$

donde W es el número de workers, $T_{work,i}$ indica el tiempo en el que el worker i-ésimo estuvo haciendo trabajo útil, $T_{up,i}$ indica todo el tiempo transcurrido en el que el worker i-ésimo estuvo activo y $T_{susp,i}$ indica el tiempo en el que el worker i-ésimo estuvo suspendido, es decir, tiempo en el que no pudo hacer ningún trabajo de la aplicación master-worker porque la máquina fue reclamada por el usuario o fue asignada a una tarea más prioritaria. La eficiencia óptima es E = 1.

El tiempo de ejecución (TE_n) se define como el tiempo transcurrido desde el inicio de la aplicación hasta su finalización, usando W workers: $TE_n = T_{finish,n} - T_{begin,n}$, donde $T_{finish,n}$ indica el tiempo en el que la aplicación finalizó cuando se usaron W workers y $T_{begin,n}$ indica el instante de tiempo en el empezó la ejecución.

El problema de gestionar aplicaciones master-worker sobre clusters ha sido abordado recientemente en la literatura por diversos entornos que permiten el desarrollo de aplicaciones paralelas adaptivas ejecutándose sobre clusters distribuidos. Entre ellos podemos citar a NetSolve [CKP+99], Nimrod [ASG+95] y AppLeS [SWB98]. NetSolve y Nimrod proporcionan una interfaz para crear aplicaciones que pueden ser descompuestas en un simple lote de tareas. En consecuencia, sus aplicaciones no pueden utilizar ningún tipo de información histórica sobre el comportamiento de la aplicación y sus políticas de gestión se limitan a conseguir tantos workers como sea posible y asignarles las tareas dinámicamente. El sistema AppLeS (Application-Level Scheduling) se centra en el desarrollo de agentes de gestión para aplicaciones paralelas aunque el diseño de cada agente se hace para una aplicación específica según los requerimientos de esa aplicación descritos en forma de modelo sintético por el propio usuario.

Existen también en la literatura otros trabajos que han analizado la forma en la que se puede usar el conocimiento sobre las características que exhibe una aplicación paralela para gestionar la asignación de las tareas que la componen [BG96]. Esos estudios se centran, sin embargo, en

sistemas multiprogramados de multiprocesadores donde el objetivo básico es la minimización del tiempo promedio de respuesta del sistema. Aunque los principios y los objetivos que guiaban estos trabajos no coinciden con los adoptados en el presente trabajo, sus resultados experimentales confirman el hecho de que las aplicaciones paralelas de tipo iterativo suelen presentar un comportamiento regular, lo que las hace susceptibles de ser gestionadas eficientemente por una estrategia dinámica que haga uso del conocimiento sobre ese comportamiento.

3. 2 Política de gestión propuesta

Por lo descrito hasta el momento, el tipo de aplicación master-worker que consideramos presenta un grado de predictibilidad elevado y, en consecuencia, es posible aprovechar esa característica para decidir la asignación de las tareas a los workers. La concepción de la estrategia propuesta en este trabajo se ha llevado a cabo teniendo en cuenta que debe exhibir las siguientes características fundamentales: dinámica, adaptativa y simple. Debe tomar sus decisiones en tiempo de ejecución sin necesitar ningún conocimiento a priori de la aplicación (dinámica); debe ser capaz de adquirir información sobre el comportamiento de la aplicación, y tomar sus decisiones de acuerdo con esa información (adaptativa); y debe ser una política con una complejidad computacional baja, con objeto de introducir la menor sobrecarga (*overhead*) posible sobre el sistema.

Nuestra estrategia consiste en un método heurístico que mantiene el tiempo promedio de ejecución de cada una de las tareas. Este promedio se calcula a partir de los tiempos de ejecución individuales consumidos por cada tarea en las distintas iteraciones de la aplicación. Una vez finalizadas todas las tareas de una determinada iteración, la lista con todos los tiempos promedios se ordena de forma decreciente y se usa ese orden para asignar las tareas en la próxima iteración. Lógicamente, en la primera ejecución la lista de tiempos promedio está vacía, por lo que la asignación de las tareas sigue un orden aleatorio. De acuerdo a los dos criterios usados en la asignación de tareas denominamos a nuestra estrategia como *Random & Average.*

4. Estudio experimental por simulación

En esta sección, compararemos diferentes estrategias de gestión con objeto de evaluar su calidad por lo que se refiere a la eficiencia y a los tiempos de ejecución obtenidos cuando se usan para gestionar aplicaciones master-worker. Nuestro estudio comparativo incluye por un lado, la estrategia propuesta *Random & Average*. Junto a ella se van a evaluar tres estrategias más que permiten cubrir diferentes aspectos relevantes por lo que se refiere a la precisión del conocimiento que usan y a su dinamicidad. De este estudio experimental nos interesa obtener información que nos permita evaluar cuál es el comportamiento promedio de la política de gestión *Random & Average* y qué posibles cotas se pueden fijar en los peores casos. Por estas razones, nuestras simulaciones se han llevado a cabo de forma sistemática variando el número de workers entre 1 y N, siendo N el número de tareas, y considerado que el conjunto de workers estaba presente durante todo el tiempo de ejecución de la aplicación (es decir, $T_{susp,i} = 0$ en ec. 1).

4.1 Descripción de las estrategias simuladas

El conjunto de estrategias utilizadas en la comparación fue el siguiente:

- **LPTF (Largest Processing Time First)**: Esta política en cada momento asigna la tarea pendiente con un mayor tiempo de ejecución. Antes de la iteración las tareas son ordenadas en forma decreciente en función del tiempo de ejecución que van a tardar en dicha iteración. Cada vez que un worker finaliza, el proceso master le asigna la siguiente tarea pendiente dentro de la lista. Es conocido que la política *LPTF* es capaz de obtener un tiempo de ejecución que no será peor que 4/3 del tiempo óptimo [Hall97]. El inconveniente de esta política reside en que

necesita tener anticipadamente una información exacta del tiempo de ejecución de las tareas, lo que hace que sea una política que no es susceptible de ser utilizada en la práctica. Sin embargo, en nuestro estudio la vamos a utilizar como una cota superior del rendimiento alcanzable por las otras estrategias.

- **LPTF on Expectation**: Esta política funciona como *LPTF*, pero las tareas se ordenan inicialmente de forma decreciente en función del tiempo de ejecución esperado. En cada iteración las tareas se asignan siguiendo ese orden. A diferencia del caso anterior la lista de las tareas es estática durante toda la ejecución. Si los tiempos de ejecución concretos de las tareas en cada iteración no varían respecto al tiempo esperado entonces el comportamiento de la política coincide con el de LPTF. Esta política sería aplicable cuando el usuario conoce de forma bastante precisa su aplicación y fija un orden inicial de las tareas en función del comportamiento que él espera de la aplicación aunque durante la ejecución de la aplicación él no pueda hacer predicciones sobre las variaciones que experimentarán los tiempos de ejecución de las tareas. En consecuencia, esta política es estática y no adaptativa.

- **Random**: Cuando un worker finaliza su trabajo, esta política le asigna aleatoriamente una tarea de la lista de tareas pendientes. Esta política se corresponde con el caso de un método puramente dinámico y no adaptativo, donde la política no tiene ningún conocimiento sobre la aplicación. En principio, esta estrategia es la que introduce la menor sobrecarga aunque también será la que obtenga el peor rendimiento. En consecuencia, la usaremos como cota inferior del rendimiento alcanzable por las demás estrategias.

4.2. Marco de simulación

Las cuatro políticas de gestión descritas anteriormente fueron simuladas sistemáticamente con objeto de evaluar su rendimiento en la gestión de aplicaciones master-worker. Los índices de rendimiento que se evaluaron fueron la eficiencia y el tiempo de ejecución (de acuerdo con las definiciones presentadas en la sección 3.1). Las aplicaciones master-worker fueron creadas sintéticamente y se escogieron cinco parámetros básicos para generar el conjunto de aplicaciones sintéticas de forma que pudiesen representar cualquier aplicación master-worker real. Los parámetros que definen las distintas aplicaciones master-worker sintéticas son los siguientes:

- *Distribución de trabajo(W)*: Este parámetro refleja cómo está repartido el trabajo entre las distintas tareas. Así hemos considerado desde situaciones donde el trabajo a realizar está repartido equitativamente entre todas las tareas (su tiempo de ejecución será parecido), a situaciones donde unas pocas tareas realizan la mayoría del trabajo y el resto se reparte entre las demás. Para ello, cada ejemplo sintético se ha creado tomando en cuenta el 20% de tareas que realizaban más trabajo y haciendo que la proporción de trabajo que realizaban fuese de un 20%, 30%, 40%, 50%, 60%, 70%, 80% o 90% (desde situaciones donde el trabajo está repartido de forma muy equitativa, hasta casos extremos donde muy pocas tareas hacen la mayoría del trabajo). Además de tener en cuenta la proporción de trabajo realizado por el 20% de tareas más cargadas y el 80% restante, las tareas de cada uno de esos dos grupos se generaron de forma que todas las tareas de un grupo tuviesen tiempos iguales o distintos.

- *Iteraciones (L)*: este parámetro indica el número de lotes de tareas que van a ejecutarse. Se consideraron valores de 10, 35, 50 y 100 iteraciones.

- *Variación (D)*: a los tiempos originales de las tareas generados según los criterios señalados en el punto de distribución del trabajo se les aplicaba en cada iteración una cierta variación para reflejar las diferencias en tiempo de ejecución que suelen exhibir las tareas cuando manejan datos distintos en cada una de las iteraciones. Los porcentajes de variación que se consideraron fueron de 0%, 10%, 30%, 60% y 100%. En el casos de una variación del 0%, los tiempos de

cada tarea eran los mismos en todas las iteraciones. Las variaciones del 10% y del 30% representan los casos de aplicaciones regulares donde existe una variación moderada en los tiempos de ejecución en función de los datos. Las variaciones del 60% y del 100% suponen que la aplicación tiene un comportamiento muy irregular y poco predecible donde los tiempos de las tareas tienen cambios muy significativos entre distintas iteraciones. Aunque estos casos se antojan poco probable en la realidad, se utilizaron para evaluar el comportamiento de las estrategias ante un caso extremo.

- *Número de Tareas (T):* Se consideraron ejemplos de aplicaciones con 30, 100 y 300 tareas, lo que suponía considerar casos con un número de tareas bajo, medio y elevado, respectivamente.

4.3. Resultados de las simulaciones

Aunque el número de simulaciones que se ha llevado a cabo es muy extenso, las principales conclusiones obtenidas se pueden ilustrar usando básicamente algunos ejemplos correspondientes a los casos en los que se usaban 30 tareas. El análisis que presentamos a continuación muestra qué influencia tiene cada uno de los parámetros mencionados en la sección anterior en el rendimiento de las políticas de gestión. En todas las gráficas que se mostrarán, el eje X contiene el número de workers usado y el eje Y contiene la eficiencia o el tiempo de ejecución obtenido por las diferentes estrategias. Los valores que encabezan las gráficas describen los valores de los distintos parámetros mencionados anteriormente y se comentarán en el punto apropiado.

- *Efecto en la eficiencia de la distribución del trabajo (W)*: La forma en la que está distribuido el trabajo entre las tareas determina el número ideal de workers con el que se obtiene la mejor eficiencia. Si el trabajo está repartido de forma más equitativa (fig 4.1a, con distribución del 30%) el número de workers con el que se puede llegar a obtener una buena eficiencia es mayor (17 en este caso). Cuando la mayoría del trabajo está concentrado en pocos workers (fig 4.1b, con distribución del 60%), el número ideal de workers es menor (11) pues basta con unos cuantos workers que hagan las tareas mayores y algún worker adicional para hacer las otras tareas de tiempo poco importante. Si la distribución es mayor, también se observa una caída más pronunciada en la eficiencia a partir del número ideal de workers (en el ejemplo, la eficiencia llega a caer hasta un valor de 0,4 con una distribución del 60%, mientras que se reduce sólo a un 0,6 con una distribución del 30%).

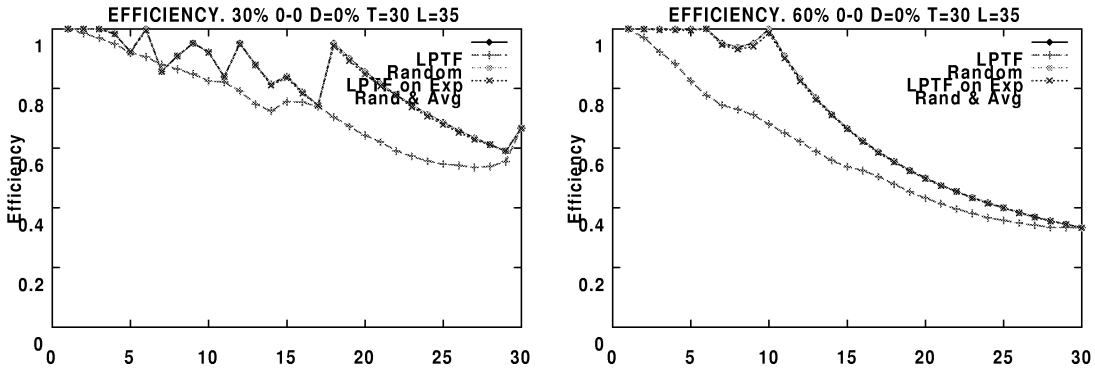

Figura 4.1. Efecto de la distribución del trabajo. (a) W=30% (b) W=60%

- *Efecto en la eficiencia del número de iteraciones (L)*: El número de iteraciones es un parámetro que sólo influye a la estrategia *Random & Average*, en los casos en los que los tiempos variaban entre diferentes iteraciones. En principio, se puede esperar que el rendimiento de la estrategia *Random & Average* sea mejor cuantas más iteraciones haga la aplicación, ya que eso le permite "aprender" mejor cual es el tiempo promedio de las tareas. Los resultados de

nuestras simulaciones muestran que aunque el número de iteraciones sea muy elevado las mejoras no son significativas. La ganancia en tiempos de ejecución y en eficiencia nunca fueron superiores al 8% entre casos con pocas iteraciones (10) y casos con más iteraciones (35 o más). Al aumentar el número de iteraciones a 50 o 100, no se observaron diferencias apreciables respecto al rendimiento obtenido con 35 iteraciones. Este dato indica, en definitiva, que la estrategia *Random & Average* necesitará de pocas iteraciones para gestionar eficientemente las tareas.

- *Efecto en la eficiencia del tamaño de las tareas*: Atendiendo al tamaño relativo de las tareas, se observa que el hecho de que el 20% de las tareas mayores sean iguales o diferentes no tiene ningún efecto apreciable. Sin embargo, el tamaño relativo del 80% de tareas restantes sí muestra una influencia importante. Así, si ese 80% está formado por tareas similares, la curva de eficiencia tiene una forma muy irregular con numerosos altos y bajos hasta llegar a el punto donde se inicia el descenso regular (estas irregularidades se pueden ver en la figura 4.1a hasta llegar a 17 workers). Este hecho implica que si el número de workers asignado a una aplicación está en esa zona baja pueden darse situaciones contradictorias donde asignar un nuevo worker pueda suponer tanto una mejora en la eficiencia como un empeoramiento, en función de las características particulares de ese grupo del 80% de las tareas menores. Cuando este grupo de tareas tienen tamaños muy similares aumentar o disminuir un worker puede suponer que se consiga o se deje de conseguir un encaje de tareas en el que todos los procesadores consuman un tiempo de cómputo similar. Si esas tareas exhiben tamaños diferentes (figura 4.2) siempre es más simple encontrar esa combinación de tareas que consiguen un encaje perfecto aunque el número de procesadores aumente o disminuya ligeramente, lo que supone que la curva de eficiencia sea más suave hasta llegar al punto de descenso contínuo.

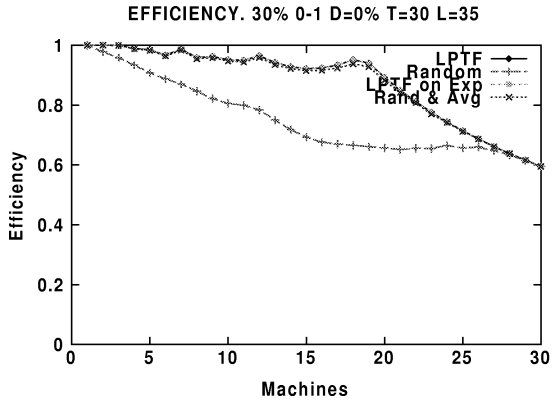

Figura 4.2. Efecto del tamaño de las tareas: 20% de tareas grandes de tamaño similar, resto de tamaño distinto

- *Efecto en la eficiencia de la variación (D)*: Como era de suponer, una política como *Random & Average*, que usa los tiempos promedios para decidir la asignación de tareas en la siguiente iteración, debe ser sensible al grado de predictibilidad y a la regularidad del comportamiento de la aplicación. Sin embargo, nuestros experimentos muestran que la pérdida en eficiencia que sufre nuestra estrategia es poco significativa incluso ante la presencia de porcentajes elevados de variación del 60% o del 100%. Cuando la variación en los tiempos de ejecución llega a ser del 100%, la pérdida de eficiencia nunca supera el 10% respecto a la eficiencia obtenida con una variación entre el 0% y el 30%.

Por lo que al tiempo de ejecución respecta, la figura 4.3 muestra cuál es el comportamiento que en general han obtenido todas las estrategias. Las dos gráficas muestran la diferencia relativa respecto a la estrategia LPTF en los tiempos de ejecución obtenidos por las otras estrategias. La figura 4.3a corresponde a un caso en el que el 20% de tareas mayores ejecutaban el 30% del trabajo y donde la variación en los tiempos de ejecución entre distintas iteraciones era de 0. Como se ve en este caso, tanto *Random & Average* como *LPTF on Expectation* obtienen unos tiempos de ejecución iguales a LPTF para cualquier número de workers. Por el contrario, la política Random obtiene unos tiempos de ejecución significativamente peores, llegando a ser cerca de un 35% peor. La figura 4-3b muestra la misma información cuando las tareas tenían una variación del 100%. Incluso en este caso extremo donde la capacidad de predicción de *Random & Average* está más limitada, su rendimiento

nunca fue peor del 10% respecto a LPTF. La estrategia *LPTF on Expectation*, que conoce la distribución original de tiempos sobre los que se aplica la variación, obtuvo en general resultados ligeramente mejores a los de *Random & Average*. Mientras que la estrategia *Random* mantuvo pérdidas que llegaron al 30%.

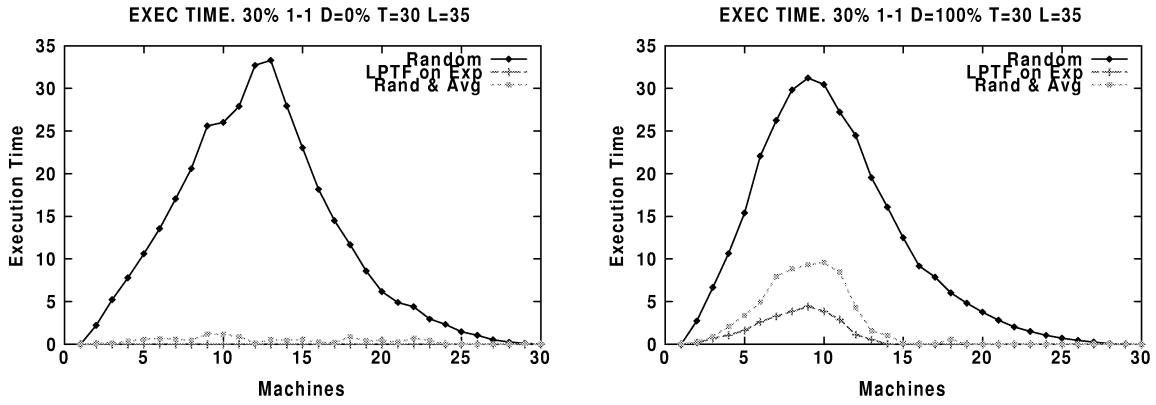

Figura 4.3. Tiempos de ejecución. (a) D=0% (b) D=100%

Es importante notar, que existe un cierto número de *workers* a partir del cual, tanto las políticas LPTF, *LPTF on Expectation* y *Random & Average* tienen un comportamiento prácticamente igual en todos los casos. Esta situación aparece cuando el número de workers asignado está por encima del máximo paralelismo que exhibe la aplicación. Por encima de ese número de workers, cualquier estrategia que vaya asignando primero las tareas mayores y las menores al final (aunque pueda cometer pequeños errores en la identificación de las tareas mayores y menores), obtiene un tiempo de ejecución equiparable al obtenido por LPTF. Tan sólo la política *Random* que no realiza ninguna selección especial de tareas mantiene tiempos de ejecución peores con un número elevado de procesadores.

4.4. Discusión

En este apartado vamos a resumir las principales conclusiones que se pueden derivar de nuestro estudio por simulación. A tenor de los resultados obtenidos podemos afirmar que la política *Random & Average* es una estrategia prometedora para gestionar la gestión de tareas de aplicaciones master-worker. La diferencia de eficiencia obtenida por la estrategia *Random & Average* respecto a la estrategia LPTF no sobrepasó el 8,65% en un 95% de los casos, llegándose en el peor caso a una pérdida del 16,86% (aunque este caso se corresponde con las situaciones poco probables en la realidad donde las aplicaciones pueden tener una grado de variación de hasta el 100% en sus tiempos de ejecución). Por lo que al tiempo de ejecución se refiere, en un 95% de los casos *Random & Average* nunca fue peor del 4% respecto a LPTF, y se llegó a diferencias de sólo el 9% en los casos patológicos donde la variación en los tiempos de ejecución eran del 100%.

Por su parte, la política *LPTF on Expectation* obtuvo resultados muy similares a *Random & Average*, tanto en lo que respecta a eficiencia como en tiempos de ejecución (en un 95% de los casos su eficiencia no fue peor del 8,91% respecto a LPTF y los tiempos de ejecución nunca fueron peor del 4%). Estos resultados relativos a *LPTF on Expectation* confirman la posible utilidad que puede suponer para un usuario tener control sobre el orden en el que deben ejecutarse las tareas de la aplicación si tiene un conocimiento relativamente preciso del comportamiento de la aplicación (lo que equivaldría en nuestras simulaciones a conocer la función de distribución de tiempos de ejecución de las tareas). Realizar la asignación según ese orden garantiza una rendimiento equiparable al de la política LPTF a pesar de que los tiempos de ejecución de las tareas se vean afectados por variaciones a lo largo de la ejecución.

De todas las estrategias probadas, la estrategia Random fue la única que obtuvo resultados significativamente peores. La pérdida de eficiencia respecto a LPTF fue en muchos casos superior al 20% llegando al 26,96 en los peores casos) y el tiempo de ejecución también fue en muchos casos entre un 20% y un 25% peor al de la política LPTF.

Por último, es importante destacar que todos los casos probados demostraron tener un intervalo de workers donde las estrategias LTPF, *LPTF on Expectation* y *Random & Average* obtenían una elevada eficiencia y un tiempo de ejecución muy parecido al que se conseguía usando el máximo número de workers (al que nunca superaron en más de un 10%).

5. Estudio experimental sobre un sistema distribuido

En esta sección vamos a presentar los resultados obtenidos con un primer prototipo de estrategia de gestión diseñada para controlar la ejecución de aplicaciones master-worker sobre clusters de estaciones de trabajo oportunísticas. Nuestro prototipo se denomina SASBATT (*Self-Adaptive Scheduling Based on Averaging Task Times*) y supone una adaptación práctica de la estrategia *Random & Average* de acuerdo con los principales resultados obtenidos en su evaluación por simulación que hemos presentado en la sección anterior.

La estrategia SASBATT realiza la asignación de tareas a workers usando una lista de tareas ordenadas decrecientemente en función de su tiempo promedio de ejecución. Este comportamiento coincide con el descrito anteriormente para *Random & Average*. Además de este control en la asignación de tareas a workers, SASBATT gestiona dinámicamente el número de workers utilizados por la aplicación. Para ello utiliza la siguiente heurística. Inicialmente, reserva tantos workers como tareas tenga la aplicación (N). Una vez completado el primer lote de tareas, se divide el tiempo total de ejecución de todas las tareas por el de la tarea mayor. El valor obtenido (denominado *speedup(N)*), refleja el *speedup* que la aplicación puede conseguir y marca también el número mínimo de workers necesarios para conseguirlo. De acuerdo con los resultados obtenidos en nuestras simulaciones, este valor está cercano al punto en el cual la curva de eficiencia está cercana a un punto máximo a partir del cual iniciará un declive paulatino hasta llegar a N workers. En sucesivas iteraciones la estrategia SASBATT reduce el número de workers tomando el valor medio entre *speedup(N)* y el último valor de workers utilizado en la iteración anterior siempre que el tiempo de ejecución de la iteración actual no supere en más de un 10% el tiempo de la tarea mayor y que se obtenga una eficiencia mayor que en la iteración anterior. Si el tiempo de ejecución de la iteración supera en más de un 10% el tiempo de la tarea mayor, la estrategia considera que se está perdiendo paralelismo en la aplicación y en sucesivas iteraciones se recuperan de uno en uno los workers liberados en la última iteración. En este caso, el límite superior que se usa es el último número de workers a partir del cual se hizo la última reducción.

Hemos realizado nuestros experimentos sobre un cluster de estaciones de trabajo de la Universidad de Wisconsin-Madison. Nuestras aplicaciones Master-Worker se desarrollaron usando la librería MW [GKL+00] que proporciona los servicios necesarios para desarrollar fácilmente este tipo de aplicaciones sobre entornos oportunísticos. En nuestro caso MW utilizaba a su vez los servicios del sistema Condor [LBR+97] que permiten, entre otras opciones, pedir y detectar los recursos computacionales disponibles en un cluster, obtener información sobre los recursos y detectar la pérdida de algún recurso. Nuestro ejemplo era una aplicación master-worker con 28 tareas cada una de las cuales calculaba una sucesión de Fibonacci. La longitud de la sucesión se fijaba aleatoriamente de forma que el tiempo de ejecución de una misma tarea en iteraciones sucesivas variaba aproximadamente un 30% y la distribución del trabajo también era del 30%.

La aplicación se ejecutó usando un número fijo de workers y usando un número dinámico de workers adaptado mediante la estrategia SASBATT. Los números de workers fijos que se probaron fueron n=28, n=25, n=20, n=15, n=10, n=5 y n=1. La tabla 5.1 muestra la eficiencia y el tiempo de

ejecución (en segundos) obtenidos en promedio en tres ejecuciones de la aplicación bajo las condiciones mencionadas anteriormente. Los resultados de la estrategia SASBATT se muestran en negrita. Como se puede ver, SASBATT fue capaz de adaptar dinámicamente el número de workers iniciales y reducirlo de 28 a 18. Gracias a la acción de SASBATT las aplicaciones se ejecutaron sobre el sistema con una eficiencia de 0,78 a la vez que se conseguía un *speedup* del 14,68; lo que supone que se estuvo muy cerca de conseguir el máximo *speedup* alcanzado usando 28 workers (aunque en este caso la eficiencia era sólo del 0,55).

En la realización de nuestros experimentos se observó en determinados casos el efecto que tenía sobre la ejecución de la aplicación la naturaleza oportunística de los recursos. Este fenómeno explica el hecho de que no siempre se consiguió reducciones en el tiempo de ejecución con un mayor número de workers, debido a los efectos producidos por la pérdida de workers y la necesidad de reasignar las tareas perdidas (casos de 20 y 25 workers en la tabla 5.1).

#Workers	1	5	10	15	**18**	20	25	28
Eficiencia	1	0,85	0,85	0,87	**0,78**	0,72	0,59	0,55
Tiempo Ejec.	36102	9269	4255	3027	**2459**	2710	2794	2434
Speedup	1	3,89	8,48	11,93	**14,68**	13,32	12,92	14,83

Tabla 5.1. Eficiencia y tiempo de ejecución de una aplicación master-worker sintética (resultados de SASBATT en negrita)

6. Conclusiones y Líneas Abiertas

En este trabajo, hemos presentado un análisis del problema relacionado con la gestión de aplicaciones master-worker ejecutadas sobre clusters de máquinas homogéneas. Hemos propuesto una política de gestión simple y adaptativa, denominada SASBATT, que toma medidas del tiempo de ejecución de las distintas tareas en las que se descompone la aplicación. Las tareas se ordenan de forma decreciente en función del tiempo promedio de ejecución de acuerdo de cada una de ellas y se asignan a los workers siguiendo ese orden. Además, la estrategia adapta dinámicamente el número de workers reservados por la aplicación para evitar que ciertos workers queden inactivos al finalizar su tarea y deban esperar a que otros workers finalicen su trabajo. Esto se hace intentando mantener, a su vez, un tiempo de ejecución cercano al tiempo de ejecución conseguido con un número de workers igual al número de tareas.

El diseño de la estrategia se ha basado en un extenso estudio a partir de simulaciones donde se comprobó que realizar la asignación de las tareas de la forma descrita permite obtener una eficiencia y unos tiempos de ejecución muy próximos a los obtenidos mediante una estrategia como LPTF que requiere conocimiento a priori sobre la aplicación. El buen comportamiento de nuestra estrategia está garantizado aún en los casos en los que la aplicación muestre un comportamiento irregular donde los tiempos de las tareas tengan oscilaciones importantes a lo largo de la ejecución de la aplicación. Un prototipo preliminar de SASBATT se ha probado experimentalmente utilizando el sistema Condor y la librería MW para crear y gestionar los recursos oportunísticos de un cluster de estaciones de trabajo. Nuestro prototipo consiguió, en general, eficiencias cercanas al 80% y un speedup próximo al máximo exhibido por la aplicación.

Existen tres líneas básicas en las que este trabajo puede extenderse. Primero sería necesario adaptar la estrategia de forma que incorporen mecanismos que minimicen en la medida de lo posible los efectos de la suspensión o la pérdida de workers. Al ser el sistema oportunístico la probabilidad de perder alguna máquina durante la ejecución de cada una de las iteraciones de la aplicación no es negligible. Aunque el sistema pueda proporcionar posteriormente máquinas que sustituyan a las que se han perdido, el efecto de la pérdida de una máquina durante una iteración puede hacer que el

tiempo de finalización de la iteración se alargue de forma ostensible a la vez que la eficiencia cae bruscamente. Técnicas como la duplicación de tareas o la reserva de máquinas adicionales deben ser estudiadas para evaluar su capacidad de aliviar los efectos de esa pérdida de máquinas.

La segunda línea de trabajo se centra en adaptar la estrategia para que tenga en cuenta la heterogeneidad de las máquinas. En este caso, asignar las tareas más costosas a las máquinas más rápidas aparece como la elección intuitivamente más razonable. Sin embargo, es preciso evaluar de qué forma se pueden aprovechar las máquinas más lentas con objeto de que también ellas contribuyan a la resolución de la aplicación sin que eso suponga que las máquinas rápidas deban permanecer inactivas esperando a que las lentas terminen.

Por último, la tercera vía de estudio se abre hacia la gestión de aplicaciones que se ejecutan sobre clusters distantes físicamente donde el ancho de banda de la red de interconexión es significativamente inferior al ancho de banda de un cluster local. En este caso, las soluciones abiertas a estudio incluye la utilización de técnicas de empaquetado de más de una tarea para su ejecución en workers distantes o la jerarquización de la propia aplicación master-worker, permitiendo que workers distantes se conviertan a su vez en masters.

7. Referencias

[ASG+95] D. Abramson, R. Sosic, J. Giddy and B. Hall, "Nimrod: a tool for performing parameterised simulations using distributed workstations", Symposium on High Performance Distributed Computing, Virginia, August, 1995.

[BG96] T. B. Brecht and K. Guha, "Using parallel program characteristics in dynamic processor allocation policies", Performance Evaluation, Vol. 27 and 28, pp. 519-539, 1996.

[BRL99] J. Basney, B. Raman and M. Livny, "High throughput Monte Carlo", Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing, San Antonio Texas, 1999.

[Buy99] R. Buyya (ed.), "High Performance Cluster Computing: Architectures and Systems", Volume 1, Prentice Hall PTR, NJ, USA, 1999.

[GKL+00] J.-P. Goux, S. Kulkarni, J. Linderoth, M. Yoder, "An enabling framework for master-worker applications on the computational grid", Tech. Rep, Univ. of Wisconsin – Madison, March, 2000.

[CKP+99] H. Casanova, M. Kim, J. S. Plank and J. Dongarra, "Adaptive scheduling for task farming with Grid middleware", International Journal of Supercomputer Applications and High-Performance Computing, pp. 231-240, Volume 13, Number 3, Fall 1999.

[GF96] V. Govindan and M. Franklin, "Application Load Imbalance on Parallel Processors", in Proc. of the Int. Paral. Proc. Symposium (IPPS'96), 1996.

[Hall97] L. A. Hall, "Aproximation algorithms for scheduling", in Dorit S. Hochbaum (ed.), "Approximation algorithms for NP-hard problems", PWS Publishing Company, 1997.

[LBR+97] M. Livny, J. Basney, R. Raman and T. Tannenbaum, "Mechanisms for high throughput computing", SPEEDUP, 11, 1997.

[PL96] J. Pruyne and M. Livny, "Interfacing Condor and PVM to harness the cycles of workstation clusters", Journal on Future Generations of Computer Systems, Vol. 12, 1996.

[SB99] L. M. Silva and R. Buyya, "Parallel programming models and paradigms", in R. Buyya (ed.), "High Performance Cluster Computing: Architectures and Systems", Prentice Hall , USA, 1999.

[SWB98] G. Shao, R. Wolski and F. Berman, "Performance effects of scheduling strategies for Master/Slave distributed applications", TR-CS98-598, University of California, San Diego, September 1998.