Entorno de Desarrollo y Sintonización de Aplicaciones Master/Worker

Paola Caymes-Scutari, Anna Morajko, Eduardo César, José G. Mesa, Genaro Costa, Tomàs Margalef, Joan Sorribes, Emilio Luque

Departamento de Arquitectura de Ordenadores y Sistemas Operativos,
Universitat Autònoma de Barcelona, 08193 Bellaterra (Barcelona) España

{paola, genaro}@aomail.uab.es
{Anna.Morajko, Eduardo.Cesar, Tomas.Margalef, Joan.Sorribes, Emilio.Luque}@uab.es

Resumen

La programación paralelo/distribuida es una tarea compleja que requiere un alto grado de experiencia para poder cubrir las expectativas de alto rendimiento computacional. El paradigma Master/Worker es uno de los paradigmas más comúnmente utilizados dado que es simple de entender y se adapta a un amplio rango de aplicaciones. Sin embargo, para obtener un rendimiento adecuado es menester sintonizar ciertos parámetros tales como la distribución de datos y el número de workers. En muchos casos, estos parámetros no pueden sintonizarse estáticamente dado que dependen de las condiciones particulares de cada ejecución. En este artículo, presentamos un entorno de sintonización dinámica que permite la adaptación de las aplicaciones a las condiciones dinámicas de ejecución, basado en un modelo teórico de comportamiento de aplicaciones Master/Worker. El entorno incluye un framework de desarrollo de aplicaciones Master/Worker, que permite al usuario concentrarse en el diseño de la aplicación, y una herramienta de monitorización/sintonización dinámica capaz de superar los cuellos de botella asociados a este tipo de aplicaciones.

Palabras clave: sintonización dinámica, sintonización automática, análisis de rendimiento, modelo de rendimiento.

Workshop de Procesamiento Distribuido y Paralelo (WPDP)

1. Introducción

El procesamiento paralelo/distribuido constituye un método muy prometedor para alcanzar altas prestaciones computacionales. Sin embargo, el uso de estos sistemas conlleva diversos aspectos que en el procesamiento secuencial clásico no son considerados:

- El diseño y desarrollo de aplicaciones paralelas implica el estudio y desarrollo de nuevos algoritmos paralelos para resolver el problema en cuestión.
- La programación de estos algoritmos paralelos requiere el uso de alguna librería de paso de mensajes (PVM, MPI) para intercomunicar los procesos que cooperan en la aplicación. El programador debe utilizar un conjunto de nuevas primitivas y adaptar el algoritmo paralelo a las posibilidades que ofrece la librería de comunicación.
- Debe analizarse el rendimiento de la aplicación para determinar los cuellos de botella que aparezcan durante la ejecución y debe modificarse la aplicación en consecuencia para solucionar los problemas de rendimiento, y así cubrir las expectativas de alto rendimiento.
- En muchos casos, el comportamiento de las aplicaciones varía según el conjunto de datos de entrada o de la plataforma subyacente. En otros casos el comportamiento puede cambiar dinámicamente debido a la evolución de la aplicación o a características dinámicas del sistema. En tales casos las acciones de sintonización deben efectuarse durante la ejecución de la aplicación, lo cual representa un gran desafío.

En este contexto, es necesario contar con nuevas herramientas de software que asistan al usuario en la especificación de aplicaciones, ocultando los detalles de bajo nivel relativos a la comunicación y que sintonice automáticamente el rendimiento de la aplicación. Es necesario ofrecer un framework de especificación de programas de alto nivel de abstracción.

Dicho framework provee un conjunto de estructuras bien definidas de manera que el usuario seleccione la estructura a utilizar y especifique las características directamente inherentes a su aplicación particular (por ejemplo, conjunto de datos y funcionalidades). Finalmente el framework genera el código utilizando la librería de paso de mensajes apropiada. De esta manera se restringe al usuario al uso de determinadas estructuras, pero vuelve más simple la especificación. Además, dado que las aplicaciones siguen estructuras conocidas es posible desarrollar modelos de rendimiento de tales estructuras y sintonizar el rendimiento de la aplicación “on the fly”.

En este artículo, se presenta un entorno completo que permite desarrollar aplicaciones paralelo/distribuidas utilizando un framework de programación y ejecutar la aplicación bajo el control de un sistema de sintonización dinámico y automático. La Sección 2 describe el diseño del framework para el desarrollo de aplicaciones Master/Worker. La Sección 3 presenta el modelo de rendimiento diseñado para esta clase de aplicaciones. En la Sección 4 se introduce el sistema de sintonización MATE y la integración del modelo de rendimiento para aplicaciones Master/Worker. La Sección 5 presenta algunos resultados experimentales realizados con una aplicación que resuelve el problema N-Body desarrollada utilizando el framework y sintonizada con MATE. Finalmente, la Sección 6 presenta algunas conclusiones.

2. Diseño del Framework

Un framework debe cubrir los siguientes requerimientos:

- *ser genérico y flexible*. Cualquier aplicación que coincida con el patrón de diseño implementado por el framework debería poder implementarse utilizándolo.
- *su utilización debería ser simple*. Los detalles de implementación del framework deberían ocultarse al usuario para proveer una API (*Application Programming Interface*) clara y fácil de comprender.
- *ser eficiente.* Aunque el framework se integre a un entorno de sintonización automática la eficiencia del framework no debe desatenderse.
- *ser independiente de la librería de comunicación subyacente.* Las funciones de librería no deberían incluirse directamente en el código del framework.

Desde otro punto de vista el objetivo de utilizar el framework en un entorno de sintonización dinámica presenta un requerimiento adicional:

- debe diseñarse para simplificar la monitorización y modificación dinámica de las aplicaciones generadas.

El framework fue desarrollado siguiendo el paradigma de orientación a objetos para encapsular su implementación y comportamiento, dejando en manos del usuario la definición de los métodos para la resolución computacional del problema.

El diseño se basa en que para cualquier modelo de programación (Master/Worker, pipeline, Divide and Conquer, etc) existen dos tipos de elementos con funcionalidades diferentes, claramente identificables: primero, el proceso encargado de efectuar los cómputos y que tiene un comportamiento específico de acuerdo con su rol real en el paradigma, y segundo la gestión de los datos que deben intercambiarse entre tales procesos.

La estructura de clases resultante se presenta en la figura 1, donde puede observarse que para cada clase de procesos en el paradigma, el framework incluye una clase *Process class* para encapsular su comportamiento; los métodos que definan los cálculos particulares a efectuar, deben ser implementados por el usuario en la clase derivada *MyProcess class* y por ello se definen como métodos virtuales. En esta clase derivada el usuario debe codificar la computación específica de la aplicación definiendo métodos de partición de datos, computación, inicialización y finalización.

Puede observarse que el framework incluye clases que encapsulan la gestión de los datos a comunicar entre los diferentes tipos de procesos (*CommunicationManagement class*). La idea es que cada objeto de cada clase de proceso en particular tenga asociado un objeto de gestión de comunicaciones responsable de la gestión de los canales lógicos utilizados para que el proceso pueda establecer comunicaciones bidireccionales. Este objeto ofrece al usuario una interface de comunicación.

Este diseño está orientado a independizar las estructuras de datos que utiliza el usuario para realizar los cálculos de las estructuras de datos utilizadas en el intercambio de información con los otros procesos, pero manteniendo la flexibilidad de decidir cómo mapear una clase de estructura en otra.

Finalmente, para cubrir el requerimiento de independencia de la librería de comunicación, se definió una clase de comunicación que encapsula las funciones de la librería de comunicación (*Communication Class*). Esta clase ofrece una interface standard con PVM y MPI. Esta clase es

generada automáticamente por una herramienta (*CCG – Communication Class Generator*) que usa información de configuración de datos provista por el programador.

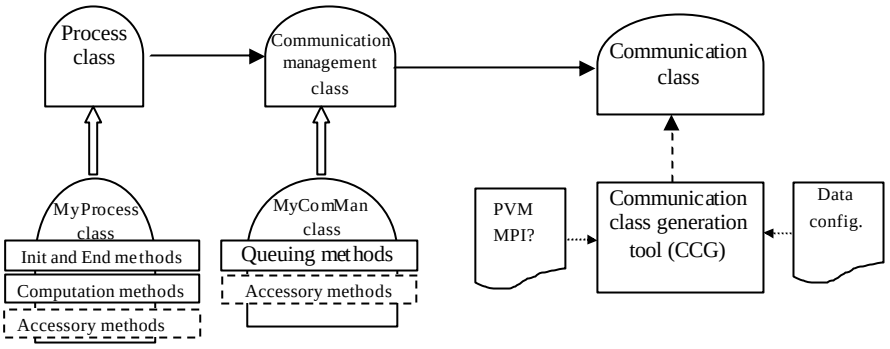

Fig. 1. *Estructura de Clases del framework*.

Implementación Master/Worker del problema N-Body

El framework se utilizó para el desarrollo de una aplicación Master/Worker que aborda el problema N-Body. Esta aplicación involucra el procesamiento de las fuerzas de interacción entre un conjunto de *n* cuerpos en el plano para cada instante de tiempo con el fin de determinar las trayectorias de los mismos. En cada instante, la nueva posición de cada cuerpo y su velocidad se calculan en función de la influencia que ejercen sobre él el resto de los cuerpos. Esta aplicación fue elegida dado que puede escalar a un gran número de workers y se ejecuta durante una serie de iteraciones correspondientes a distintos pasos de tiempo.

El framework provee una infraestructura para el desarrollo de una aplicación bajo el paradigma Master/Worker. Para desarrollar una aplicación Master/Worker el usuario debe extender las clases generales del master y del worker. Cada una de estas clases tiene métodos virtuales C/C++ que deben definirse para determinar la funcionalidad de cada actor (master o worker) dentro de la aplicación. La comunicación entre el código del usuario y el framework se realiza por medio de polimorfismo y los atributos que heredan las subclases.

El primer paso para el desarrollo de la aplicación consiste en ejecutar el script "*create*" que genera un árbol de directorios que contiene todos archivos necesarios. Crea un directorio "master" y un directorio "*worker*" que contiene el código específico del master y del worker provisto por el framework, y clases que el usuario debe rellenar. El usuario sólo debería modificar "*_mymaster.cpp*" y "*_mymaster.h*" en el directorio "*master*" y "*_myworker.cpp*" y "*_myworker.h*" en el directorio "*worker*".

Otro directorio creado es "*estructuras*", que contiene las estructuras del usuario incluyendo los archivos "*myinstruct.h*" y "*myoutstruct.h*". El usuario debería editar dichos archivos y definir las estructuras de entrada y salida respectivamente, siguiendo la sintaxis de C/C++. Una vez codificadas las estructuras de datos de entrada y de salida, el usuario debe codificar las clases del master y del worker. El framework soporta iteraciones y provee ejecución síncrona y asíncrona. Ello afecta la manera en la que el master distribuye las tareas a los workers.

La ejecución involucra una serie de eventos que debe definir el usuario. Tales eventos son la inicialización del master ("*_M_Initialize*"), inicialización del worker ("*_W_Initialize*"), partición de

datos del master (“*_M_Partition*”), cómputo del worker (“*_W_DoWork*”), acción final del worker (“*_W_FinalWork*”), recuperación de datos del master (“*_M_Recover*”) y trabajo final del master (“*_M_FinalWork*”), que deben definirse en *_mymaster.cpp* y *_myworker.cpp* según corresponda.

En la implementación de N-Body, la función “*_M_Initialize*” se utiliza para reservar la memoria necesaria para el arreglo de cuerpos, posición y velocidad iniciales. El método *_W_Initialize* se utiliza para inicializar el buffer de resultados, un arreglo para almacenar los resultados del cómputo realizado. Cada estructura de entrada es un conjunto de cuerpos en el rango de procesamiento que indica al worker el subconjunto de cuerpos que debe procesar. La generación de tareas se basa en el número de workers disponibles y la división se realiza equitativamente, donde cada worker recibe aproximadamente la misma cantidad de trabajo (“*_M_Partition*”). El método “*_W_DoWork*” del worker consiste en procesar las estructuras de entrada. Ello consiste en calcular para cada cuerpo en el rango definido la aceleración resultante en base a la masa y la distancia entre los cuerpos. La aceleración resultante modifica la velocidad, dirección y posición actual de los cuerpos. La estructura de datos de salida es una estructura que contiene la nueva posición y velocidad de los cuerpos procesados. Luego de procesar cada estructura de entrada, los resultados se almacenan en una estructura de salida y en una estructura propia de la aplicación denominada “spaces”. La colección de resultados se encolan para ser enviados en el método “*_W_FinalWork*”, y spaces queda liberado para la siguiente iteración. El método “*_M_FinalWork*” se utiliza para almacenar los resultados en un archivo.

3. Modelo de rendimiento para el Número de Workers

En esta sección se presenta el problema de determinar el número de workers apropiado para una aplicación Master/Worker. El problema se considera para aplicaciones MASTER/WORKER homogéneas, es decir aquellas aplicaciones en las que todas las tareas tienen aproximadamente el mismo tamaño y requieren un procesamiento similar. De hecho, esta clase de aplicaciones se comporta de forma similar a aplicaciones MASTER/WORKER balanceadas con el mismo tiempo total de procesamiento y el mismo volumen global de comunicación, tal como se muestra en [1].

Para este análisis, se asume la siguiente terminología para identificar los diferentes parámetros que formarán parte del modelo de rendimiento:

tl= latencia de la red, en milisegundos (ms)
l = costo de comunicar un byte (relación inversa del ancho de banda), en ms/byte
v_i= tamaño de las tareas enviadas al worker i, en bytes
v_m= tamaño de los resultados enviados por cada worker al master, en bytes
V= volumen total de datos ($\sum(v_i + v_m)$), en bytes
n= número actual de workers en la aplicación
tc_i = tiempo de cómputo del worker i, en ms
Tc= tiempo total de cómputo ($\sum(tc_i)$), en ms
Tt= tiempo total insumido por una iteración de la aplicación, en ms
$Nopt$= número de workers necesario para minimizar el tiempo de ejecución

En [1] se muestra el análisis realizado para construir las funciones de rendimiento asociadas a esta clase de aplicaciones, de las que se derivan dos expresiones diferentes que permiten calcular el número óptimo de workers en diferentes situaciones.

Inicialmente, el master debe enviar un conjunto de tareas a los workers, lo cual implica un cierto tiempo de comunicación en función de la latencia y ancho de banda de la red y del volumen de datos a comunicar. Si el protocolo de comunicación es asíncrono la latencia de la red y el tiempo de comunicación de los datos se solapan y, en consecuencia, el que sea mayor será el que tendremos en cuenta.

Así, si el protocolo de comunicación es asíncrono y *tl* es mayor que $l * v_i$, entonces el reparto de tareas tardará: $n * tl + l * v_i$. Si por el contrario, $l * v_i$ es mayor que *tl*, entonces el reparto tardará: $tl + l * \sum_{i=0}^{n-1} v_i$. Ahora bien, si el protocolo de comunicación es síncrono entonces el tiempo necesario para repartir las tareas será: $n * tl + l * \sum_{i=0}^{n-1} v_i$.

Una vez terminada la distribución de los datos, y dado que también hay solapamiento entre las comunicaciones y el cómputo, sólo debemos tomar en cuenta el tiempo de procesamiento de un worker, esto es tc_i.

Finalmente, debemos contabilizar el tiempo de respuesta de los workers al master; como seguimos teniendo solapamiento de cómputo y comunicaciones sólo debemos tomar en cuenta una comunicación, es decir: $tl + l * v_m$. La condición necesaria para que esto sea cierto es que el master esté preparado para recibir, lo cual es siempre cierto si $tl > l * vi$, pero puede no serlo si $tl \leq l * vi$. En este último caso también debe cumplirse la condición de que el tiempo de distribución de tareas debe ser mayor que el tiempo de respuesta del primer worker. Así, el tiempo necesario para completar una iteración (distribuir-procesar-responder) en una aplicación Master/Worker será la suma de estas cantidades en cada una de las circunstancias descritas, lo cual puede ser expresado matemáticamente de la siguiente forma:

$$Tt = 2 * tl + l * \sum_{i=0}^{n-1} v_i + tc_i + l * v_m$$

$$\text{si} \quad ((tl \leq l * v_i) \quad \text{y} \quad (tl + l * \sum_{i=0}^{n-1} v_i > 2 * tl + l * v_i + tc_i + l * v_m))$$

o

$$Tt = n * tl + l * v_i + tc_i + tl + l * v_m \qquad \text{si} \quad tl > l * v_i$$

En [1], se asumió que el master reparte un volumen constante (*V*) de datos entre un conjunto de *n* workers, razón por la cual al aumentar el número de workers se reduce el tamaño de los mensajes, así $v_i = p*V/n$ (una porción del total de datos) y $v_m = (1-p)*V/n$. Además, como se asume que la aplicación es homogénea, entonces $tc_i = Tc/n$.

Haciendo las sustituciones correspondientes se obtuvo el siguiente conjunto de expresiones:

$$Tt = \frac{(2 * tl + l * V * p) * n + Tc + l * (1 - p) * V}{n}$$

$$si((tl \leq l * \frac{p * V}{n}) y (n \leq \lceil (l * V + Tc) / (l * p * V - tl) \rceil)) \qquad \textbf{(1)}$$

o

$$Tt = n*tl + \frac{Tc}{n} + tl + l*\frac{V}{n} \qquad \text{si}(\ tl > l*\frac{p*V}{n}) \qquad \textbf{(2)}$$

a partir de las que pueden derivarse expresiones para el cálculo del número óptimo de workers para una aplicación; de esta forma:

Si el protocolo es síncrono y $n*(tl + l*V/n) > 2*tl + l*V/n + Tc/n$, entonces:

$$N_{opt} = \sqrt{(l*(1-p)*V + Tc)/tl} \qquad \textbf{(3)}$$

Si el protocolo es asíncrono y $(tl > l*\frac{p*V}{n})$ o $(tl \le l*p*V/n)$ y $(n \le \lceil (l*V + Tc)/(l*p*V - tl) \rceil$, entonces:

$$N_{opt} = \sqrt{(l*V + Tc)/tl} \qquad \textbf{(4)}$$

Tal como se dijo anteriormente, este modelo es válido si el master reparte un volumen constante (*p**V*) de datos entre un conjunto de *n* workers, pero en el caso del N-Body todos los datos son enviados a todos los workers y por tanto el volumen de datos a comunicar no es constante sino que depende del número de workers. Entonces, nótese V_i el volumen de datos que envía el master a un worker (magnitud fija, independiente de la cantidad de workers) y V_m al volumen total de resultados que envían los workers al master.

En este caso (1) y (2) se transforman en:

$$Tt = \frac{(2*tl + l*n*V_i)*n + Tc + l*V_m}{n} \qquad \textbf{(5)}$$

y

$$Tt = n*tl + l*V_i + \frac{Tc}{n} + tl + l*\frac{V_m}{n} \qquad \textbf{(6)}$$

respectivamente, y (4) se transforma en:

$$N_{opt} = \sqrt{(l*V_m + Tc)/tl} \quad \text{si } (tl > l*\frac{V_i}{n}) \qquad \textbf{(7)}$$

o

$$N_{opt} = \sqrt{(l*V_m + Tc)/l*V_m} \qquad \text{si no} \qquad \textbf{(8)}$$

Puede observarse que, en cualquier caso, los parámetros que deben monitorizarse para aplicar el modelo de rendimiento asociado a una aplicación Master/Worker son:

- tl y l. Deben calcularse al comienzo de la ejecución y deberían actualizarse periódicamente para permitir que el sistema se adapte a las condiciones de carga de la red.
- Tamaño de los mensajes que el master envía a los workers y de las respuestas que recibe, para calcular la porción de volumen de datos enviada a los workers y el volumen recibido de ellos.

- El tiempo que los workers insumen para el procesamiento de las tareas, para calcular el tiempo total de cómputo *(Tc).*

4. MATE

En esta sección se presenta MATE (Monitoring, Analysis and Tuning Environment) [2, 3] que provee sintonización automática y dinámica de aplicaciones paralelo/distribuidas. En tiempo de ejecución MATE instrumenta automáticamente una aplicación en ejecución para recoger información acerca de su comportamiento. La fase de análisis recibe eventos, detecta cuellos de botella aplicando un modelo de rendimiento y determina las soluciones para paliar tales problemas de desempeño. Finalmente, la aplicación se sintoniza dinámicamente mediante la materialización de las soluciones provistas por la fase de análisis. Durante el proceso de sintonización, no es necesario recompilar, relinkear ni reejecutar la aplicación. Para modificar la ejecución de la aplicación "on the fly" MATE utiliza una técnica denominada instrumentación dinámica [4].

Los componentes de MATE cooperan entre sí controlando y procurando mejorar la ejecución de la aplicación. Los principales componentes de MATE son los siguientes:

- Application Controller (AC), una suerte de proceso demonio que controla la ejecución de la aplicación en un host determinado (gestión de procesos y máquinas). Además provee la gestión del proceso de instrumentación y modificación.
- Dynamic Monitoring Library (DMLib), una librería compartida que el AC carga dinámicamente en los procesos de la aplicación para facilitar la instrumentación y recolección de datos. La librería contiene funciones que son responsables del registro de eventos con todos los atributos requeridos por/para la etapa de análisis.
- Analyzer, un proceso que lleva a cabo el análisis de rendimiento de la aplicación; detecta los problemas de rendimiento existentes automáticamente "on the fly" y requiere las modificaciones necesarias para mejorar el rendimiento de la aplicación.

Una cuestión importante es la representación del conocimiento de los problemas de rendimiento que se utiliza para la optimización de la aplicación. En MATE, en materia de conocimiento, se utilizan los siguientes términos: *puntos de medida, modelo de rendimiento y acciones y/o puntos de sintonización*. Un punto de medida es un punto en un proceso donde puede insertarse instrumentación. Un modelo de rendimiento consiste en activar condiciones (condiciones del comportamiento de la aplicación que se consideran cuellos de botella) y/o fórmulas que modelan la aplicación, permitiendo la determinación de condiciones óptimas. Los puntos de sintonización son componentes de la aplicación que deben modificarse para mejorar el rendimiento. Las acciones de sintonización representan las acciones que deben ejecutarse en un punto de sintonización. El conocimiento requerido para representar el modelo de rendimiento de un cuello de botella en una aplicación se especifica en un componente denominado "tunlet".

Implementación de un Tunlet

Para poder sintonizar dinámicamente el número de workers, debe adoptarse una aproximación cooperativa dado que se requiere cierta información de la aplicación. Con el conocimiento provisto por el framework presentado anteriormente, se implementó un tunlet específico para la sintonización del número de workers. La aplicación es iterativa; cada proceso ejecuta todas las

operaciones repetidas veces. En cada iteración, el master distribuye tareas a un número de workers específico y espera los resultados. Los procesos workers calculan los resultados y los envían al master. Así, el comienzo de cada iteración se utiliza como punto de sincronización de la aplicación.

El tunlet que optimiza el número de workers requiere monitorización dinámica de las funciones responsables del intercambio de mensajes (métodos de envío y recepción implementados en el framework). En particular, para estos métodos se monitoriza cada entrada o salida ya sea ejecutado por el master o los workers. La instrumentación de estas funciones y la medición de la cantidad de datos enviados a los workers y recibidos por el master, el tiempo total de cómputo de los workers, el overhead de la red y el ancho de banda, constituyen las medidas requeridas por el modelo de rendimiento presentado en la Sección 3 (expresiones (6) y (8)).

A lo largo de la ejecución, la aplicación debe conocer el número actual de workers. El modelo se evalúa al final de cada iteración, cuando todas las métricas fueron recogidas. Si el número óptimo de workers calculado en función del estado corriente del sistema difiere del número actual de workers, se invoca la acción de sintonización asociada. Puede observarse, que se requiere que la aplicación esté preparada para la introducción de modificaciones. Por ello, el framework provee una variable específica que representa el número óptimo de workers. MATE cambiará dicha variable actualizando su valor automáticamente según las condiciones actuales del entorno; el nuevo valor será utilizado en la siguiente iteración. Este proceso sólo puede realizarse entre dos iteraciones dado que la redistribución del trabajo que está siendo procesado sería una tarea complicada. Una vez que el número de workers ha sido ajustado, el trabajo puede distribuirse adecuadamente a todos los workers.

Si hay necesidad de agregar workers, se necesitarán tantas máquinas (procesadores) adicionales como workers haya que agregar. Ejecutar más de un worker en una máquina no tendría sentido dado que el tiempo de CPU sería dividido entre los workers, a menos que la máquina sea un multiprocesador.

5. Resultados Experimentales

Esta sección presenta los resultados experimentales obtenidos de la aplicación del entorno de desarrollo y sintonización dinámica. Los experimentos se realizaron sobre la aplicación N-Body, en un cluster homogéneo de Pentium 4, 1.8 Ghz, (SuSE Linux 8.0) conectado por una red de 100Mb/seg. Cada experimento fue ejecutado varias veces y se calculó el tiempo de ejecución promedio.

Dada la necesidad de controlar la carga en el sistema para reproducir los experimentos varias veces, se crearon ciertos patrones de carga externa para simular un sistema de tiempo compartido. En particular, se diseñaron un patrón de carga que crece gradualmente, y otro de carga variable. La aplicación fue ejecutada con varios números fijos de workers (1, 2, 4, 8, 16, 19) y también bajo el control de MATE, caso en el que el número de workers se adaptó dinámicamente. En cada escenario cada worker se ejecutó en una máquina individual. Los experimentos fueron conducidos en dos escenarios:

- en el primer escenario, N-Body se ejecutó en diferentes cantidades fijas de workers, sin sintonización (en coejecución con el patrón de carga).

- en el segundo escenario, la aplicación fue ejecutada bajo MATE aplicando la sintonización del número de workers (también en coejecución con el patrón de carga). La aplicación comenzó con un worker, y durante la ejecución el número fue modificándose de acuerdo al modelo descrito en la sección 3. En este escenario una máquina del cluster fue dedicada a la ejecución del Analyzer para evitar sobrecargar la ejecución de la aplicación.

La tabla 1 y la figura 2 presentan los resultados experimentales. Indican el tiempo de ejecución de la aplicación considerando diferentes números de workers y el tiempo de ejecución de la aplicación bajo MATE. En este escenario la aplicación se ejecutó conjuntamente con el patrón de carga variable.

#workers	1	2	4	8	16	19
Tiempo de Ejecución	64,49	34,61	18,09	10,37	11,83	15,49
N-Body + MATE			La ejecución comienza con 1 worker y luego se sintoniza			
Tiempo de Ejecución			10,92			

Tabla 1. *Tiempo de ejecución de N-Body (en segundos) bajo el patrón de carga variable*

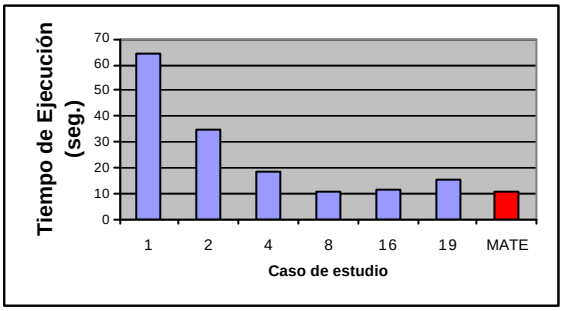

Fig. 2. *Tiempo de ejecución de N-Body utilizando diferentes cantidades de workers y MATE.*

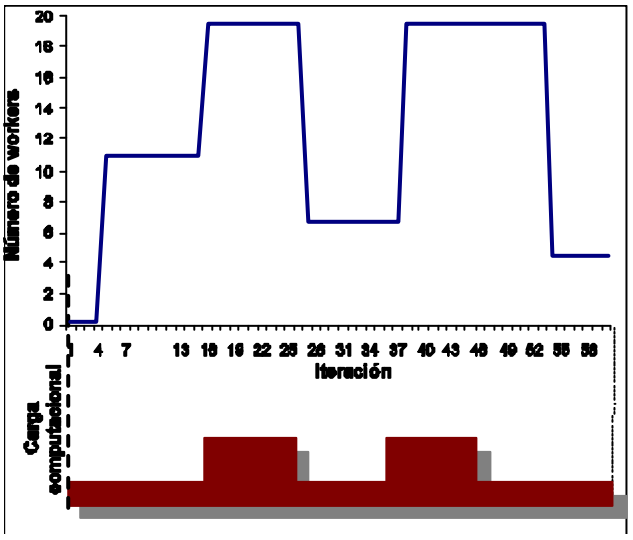

Fig. 3. *N-Body ejecutado bajo MATE. Adaptación dinámica del número de workers según el patrón de carga variable.*

La tabla 2 y la figura 4 presentan los resultados experimentales obtenidos para el patrón de carga creciente. Expresan el tiempo de ejecución de la aplicación considerando diferentes números de workers y el tiempo de ejecución de la aplicación bajo MATE. Similarmente al escenario anterior, en cada caso de estudio N-Body fue ejecutado conjuntamente con el patrón de carga creciente.

#workers	1	2	4	8	16	19
Tiempo de Ejecución	73,23	37,11	19,28	11,76	13,05	14,50
N-Body + MATE			La ejecución comienza con 1 worker y luego se sintoniza			
Tiempo de Ejecución			12,46			

***Tabla 2**. Tiempo de ejecución de N-Body (en segundos) bajo el patrón de carga creciente.*

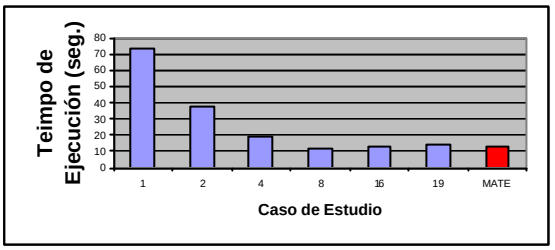

***Fig. 4**. Tiempo de ejecución de N-Body utilizando diferentes cantidades de workers y MATE.*

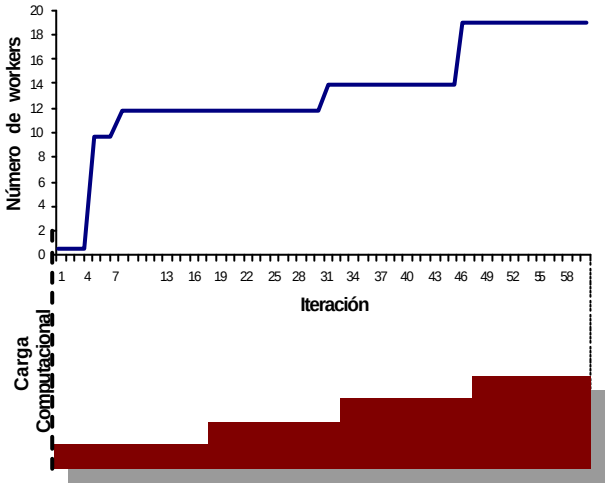

***Fig. 5**. N-Body ejecutado bajo MATE. Adaptación dinámica del número de workers según el patrón de carga creciente.*

Véase la figura 3 y la figura 5, en las que se representa la adaptación del número de workers utilizado por la aplicación a lo largo de 60 iteraciones bajo el patrón variable y creciente respectivamente. Inicialmente se ejecuta la aplicación con un único worker; los datos recolectados en las primeras iteraciones advierten la necesidad de incluir más workers para alcanzar una ejecución óptima. Posteriormente, conforme la carga del sistema aumenta o disminuye, el número de workers se adapta según corresponda, de manera que se utilice en cada momento exactamente la cantidad de workers necesaria para lograr un óptimo rendimiento.

Nótese que las respuestas a los cambios producidos en los patrones de carga se introducen algunas iteraciones posteriores (normalmente una o dos). El análisis que se realiza sobre los datos de una iteración, introduce las modificaciones pertinentes en la iteración siguiente. Es por ello que la sintonización dinámica requiere que los problemas tengan una cierta persistencia en el tiempo. En algunas ocasiones, puede necesitarse una sintonización adicional cuando los cambios de carga son considerables, dado que el tunlet puede evaluar el modelo con el número óptimo de workers estimado, y sobre la información que con ellos recoja reajustar el número óptimo. Este es el caso de la primera y segunda adaptación en la figura 5.

Puede observarse que el tiempo de ejecución de la aplicación bajo MATE es cercano a los mejores tiempos obtenidos con diferentes cantidades fijas de workers. Sin embargo, los recursos dedicados a la aplicación en cada instante utilizando MATE toman en consideración los requerimientos actuales de la aplicación y de esa manera los recursos son utilizados cuando realmente son necesarios.

6. Conclusiones

El desarrollo de aplicaciones paralelo/distribuidas eficientes puede ser una tarea dificultosa para programadores no expertos. Deben proveerse herramientas que asistan al usuario en la fase de desarrollo y ofrezcan sintonización dinámica de tales aplicaciones.

En este artículo se ha descrito el framework para el desarrollo de aplicaciones Master/Worker y la herramienta de sintonización dinámica. El framework facilita el desarrollo de la aplicación, ocultando los detalles de bajo nivel y la sintonización de rendimiento puede realizarse exitosamente "on the fly". Mediante la utilización de este entorno, los programadores pueden diseñar las aplicaciones de forma sencilla y no necesitan preocuparse por el análisis o sintonización de rendimiento, dado que la sintonización dinámica de rendimiento se ocupa de tales tareas automáticamente.

El modelo de rendimiento para evaluar el número óptimo de workers fue integrado en MATE mediante el "tunlet" correspondiente.

Se realizaron experimentos con una aplicación paralelo/distribuida desarrollada con el framework presentado y sintonizada con MATE. De esta manera se ha probado que el entorno es efectivo y su utilización beneficiosa. La ejecución de la aplicación bajo el control de MATE permitió adaptar su comportamiento a las condiciones existentes, y así mejorar su rendimiento.

7. Referencias

1. César, E., Mesa, J.G., Sorribes, J., Luque, E. "Modeling Master-Worker Applications in POETRIES". IEEE 9th International Workshop HIPS 2004, IPDPS, pp. 22-30. April, 2004.
2. Morajko, A., Morajko, O., Jorba, J., Margalef, T., Luque, E. "Dynamic Performance Tuning of Distributed Programming Libraries". LNCS, 2660, pp. 191-200. 2003.
3. Morajko, A., Morajko, O., Margalef, T., Luque, E.. "MATE: Dynamic Performance Tuning Environment". LNCS, 3149, pp. 98-107 (2004).
4. Buck, B., Hollingsworth, J.K. "An API for Runtime Code Patching". University of Maryland, Computer Science Department, Journal of High Performance Computing Applications. 2000.
5. Wilkinson, B., Allen, M.: "Parallel programming - Techniques and Applications Using Networked Workstations and Parallel Computers". Pearson Prentice Hall. Second Edition. ISBN 0 13 140563 2. 2005.