USO DE THREADS PARA LA EJECUCIÓN EN PARALELO SOBRE UNA MALLA COMPUTACIONAL

Julio Monetti & Oscar Leon /Ingeniería en Sistemas de Informacón/FR.Mendoza -UTN
jmonetti@frm.utn.edu.ar , oleon@frm.utn.edu.ar

CONTEXTO

El presente trabajo muestra el uso y análisis, desde el punto de vista de los Sistemas de Información, de metodologías no clásicas de computación a través de programación concurrente, observando características estructurales del software y hardware empleado, con el fin de asistir a aquellos usuarios que requieren la computación de alto rendimiento. Una alternativa para optimizar y compartir recursos de computación es la computación basada en ambientes *Grid* [1] . Se discute la administración de recursos a través de *Threads* en Java, para lanzar tareas sobre un *Grid*, con el objeto de paralelizar las mismas. Se presenta la arquitectura de la aplicación, diseñada sobre un modelo de 3 capas, y se comentan aspectos de su implementación, en un caso concreto para resolver un problema numérico.

Palabras clave: *paralelismo, programación concurrente.*

1. INTRODUCCION

En [2] se discutió la migración de una aplicación paralela realizada en MPI [3] sobre un *cluster*, hacia una arquitectura computacional desacoplada y con recursos descentralizados. Bajo esta migración se mantuvo el comportamiento paralelo del sistema, marcado fundamentalmente por la descomposición del dominio de datos a procesar. El nuevo ambiente de ejecución es una malla computacional montada sobre una red de área local, donde no se cuenta con las características que provee un *cluster*: paso de mensajes entre procesadores y facilidad en la sincronización de procesos que provee MPI. No obstante, la posibilidad de crecimiento se ve sumamente favorecida, ya que se trata de una arquitectura heterogénea y teóricamente sin restricciones de escalamiento. El problema de estudio es la solución de una ecuación diferencial a través del método de diferencias finitas [4]. La migración de la aplicación desde *cluster* a *Grid* presenta nuevos problemas desde el punto de vista informático y la implementación de algoritmos que soporten dichas arquitecturas. Por otro lado, se probó en [2] que el algoritmo secuencial se puede mantener sin cambios luego de la migración, si se respeta un buen diseño en la codificación. La ingeniería de software como disciplina de desarrollo, sugiere la reutilización de código tanto a lo largo de un sistema, como así también a través de diferentes sistemas. El paradigma orientado a objetos toma este principio y lo pone como uno de los principales cimientos en el diseño de grandes sistemas. Esta forma de concebir la reutilización de código se materializa a través de técnicas de programación, donde el producto final de software es concebido en términos de diferentes capas. La programación en tres capas sugiere el desarrollo en forma separada de interfaces e interacción con el usuario en la capa más alta. En otra capa (o conjunto de componentes de software) la lógica principal del sistema. Finalmente en la capa inferior todos aquellos componentes de software que toman forma de servicios a ser explotados por la capa inmediata superior. En aquellos ambientes clásicos de programación, entiendasé por tal a la

secuencial, la cual es generalmente utilizada para aplicaciones de cálculo y de carácter científico, no se aplica con frecuencia esta metodología, dando lugar a la duplicación de código, o a una mala estructuración del mismo.
Uno de los objetivos del presente trabajo, es reutilizar el código de cálculo, que será explotado en una arquitectura más enriquecida: la de computación distribuida. De esta forma se puede visualizar el producto como un conjunto de componentes de cálculo ubicados en la capa inferior o de servicios, una capa de software que conforma la lógica del sistema, y está destinada a la coordinación de los diferentes procesos de cálculo, y finalmente el *postproceso* ubicado en la capa de interface.

2. DESARROLLO

2.1 Descomposición del Dominio de Datos. Se emplea un *middleware* o capa de gestión para intercambiar información entre los equipos que constituyen el *cluster*. Este *middleware* es un componente de software que asegura una correcta administración de los procesos: sincronización, mensajes, etc. MPI permite la comunicación entre procesos residentes en diferentes computadoras de un *cluster*. El pasaje de mensajes entre ellas asegura la sincronización de las mismas y el intercambio de datos. Se debe considerar que una aplicación clásica en MPI supone distribuir el mismo programa a lo largo de todos los nodos del *cluster*, donde cada uno de estos ejecutará solo una porción del programa.

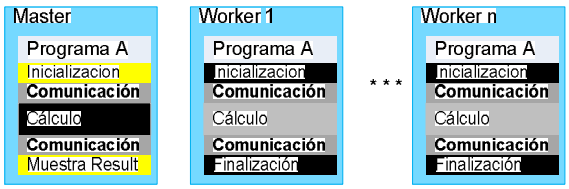

FIGURA 1. Distribución del *Programa A* al conjunto *workers*.

En la figura [1] se observa una estación de trabajo, el *master*, responsable de sincronizar el trabajo del resto de las estaciones (*workers*). El *master* es el encargado de particionar el conjunto de datos original, distribuirlo a cada *worker*, sincronizar la ejecución de las tareas de cálculo en cada uno de ellos y recibir los datos procesados. La versatilidad provista por MPI en la utilización de un *cluster* de computadoras permite el paralelismo y sincronismo de tareas llevada a cabo por cada *worker*. Como se mencionó en párrafos anteriores, con este conjunto de librerías se hace posible la ejecución del mismo programa en diferentes computadoras actuando cada uno de ellos sobre una parte diferente del dominio de datos, que en este caso está dado por el conjunto de puntos donde la ecuación diferencial debe ser calculada.

2.2. Multithreading como Solución a la Gestión de Procesos. En un entorno computacional donde existen diversos recursos de cálculo pero no la arquitectura de un *cluster*, la funcionalidad de MPI debe ser reemplazada por una arquitectura alternativa, por ejemplo una *malla computacional*, compuesta de nodos heterogéneos y sin una ubicación geográfica determinada. Se deberán emular en este caso los procedimientos de comunicación, administración y sincronización de tareas. Para satisfacer estas tareas de administración, resulta adecuada la utilización de *n* procesos ejecutándose en el *master* para mantener independientemente la administración y comunicación con los *n* procesos de cálculo, ahora distribuidos a lo largo de la malla computacional. Por otro lado, si se garantiza paralelismo entre estos *n* procesos de administración, se puede lograr un mayor paralelismo entre los nodos *workers*.
Se utiliza una aplicación escrita en lenguaje Java, residente en el *master*, y se mantienen los códigos de cálculo codificados en lenguaje *C* en cada uno de las estaciones de trabajo. Cada hilo de

ejecución, representa un conjunto de instrucciones secuenciales, las cuales son ejecutadas en forma paralela con otros hilos de ejecución, eventualmente bajo la supervisión de un hilo coordinador. Cada hilo es responsable del envío del conjunto de datos correspondiente al proceso *worker* que está administrando. En el caso de descomposición del dominio, cada hilo actúa sobre un conjunto de datos diferente. No obstante ello, se debe tener en cuenta que todos los hilos comparten los mismos recursos dentro de la misma estación de trabajo: compartiendo espacio de direcciones, acceso al disco rígido, procesador, etc. Un programa Java puede ser considerado como un hilo único de ejecución al utilizar un flujo único de control. Dicho hilo tiene la posibilidad de crear distintas instancias de control, las que serán consideradas nuevos hilos. Cuando un determinado hilo modifica un dato en memoria este puede ser transmitido a través de un mensaje a otro hilo, así emulando el comportamiento de MPI. Java oculta la complejidad en el tratamiento de múltiples instancias de ejecución tras una manera muy simple y conveniente de codificación, creando cada hilo de ejecución como un objeto de datos [5]. Estos objetos (clase *Thread*) cuentan con un estado autónomo, contador de programa, estado de la pila e información sobre el estado de CPU. Esto se ve complementado por información en forma de atributos contenidos en dicha clase de datos. Luego, cada hilo comienza y termina su trabajo de forma independiente. El trabajo de cálculo llevado a cabo por las rutinas y/o módulos numéricos se reutiliza con relativa facilidad a partir de la codificación serial en lenguaje *C*. Como se especificó anteriormente, este código ya compilado reside en cada nodo *worker* (nodo de la malla computacional), y es responsable únicamente del cálculo de la ecuación en cada punto del subdominio de datos local. Por otro lado las tareas comunes llevadas a cabo por el nodo *master* son codificadas en el hilo principal del programa administrador. El objeto Java empleado para describir una tarea en forma remota cuenta tanto con la información necesaria para ejecutar el programa en la estación de destino (identificador del recurso, tipo de estación, etc.), como así también con información necesaria para el control de ejecución de dicho trabajo. El hilo principal decide sobre la continuidad o no de determinado hilo, su recuperación, migración hacia otra estación, etc. Esto resulta sumamente ventajoso para realizar recuperación ante fallos debido a la salida de servicio de un determinado *worker* en tiempo de ejecución. El cuerpo del programa principal es encargado de crear los distintos hilos (clase *Trabajo* que hereda *Thread*), su iniciación y control. Puesto que existe una determinada cantidad de trabajos a ejecutar, cada objeto es contenido en un único arreglo (clase *Vector*).

2.3 Hilos de Ejecución y Grid Computing. Las mallas computacionales [1] permiten el aprovechamiento de recursos descentralizados, con el objeto de incrementar la capacidad de cómputo, almacenamiento de datos, etc. De esta forma, es posible presentar al usuario final el conjunto de recursos como una estación de cálculo única con gran capacidad de cómputo, requiriendo adecuar sensiblemente las aplicaciones finales. Por otro lado, un rasgo importante de un *cluster* a tener en cuenta es que este representa una herramienta dedicada y fuertemente acoplada, dedicada a resolver problemas de alto rendimiento y, en general, de alta disponibilidad, lo que conlleva una inversión importante en recursos de *hardware* e instalaciones auxiliares. Las mallas computacionales utilizan recursos provistos por equipos distribuidos geográficamente y conectadas por una red para resolver problemas de computación de gran escala. Estas posibilitan la realización de cómputos sobres grandes conjuntos de datos, los que se subdividen en partes más pequeñas, siendo estos procesados en los nodos que constituyen la *Grid*. En este sentido se requiere, al igual que en el caso

de c*lusters*, que la aplicación presente concurrencia y en este caso también se aprovecha el paralelismo para realizar cálculos simultáneos. Así, la malla provee la capacidad de realizar computación de alto rendimiento, modelando una arquitectura de computadora virtual que permite, entre otras cosas, distribuir la ejecución de procesos a través de una estructura computacional enriquecida. Cada computadora disponible en el laboratorio, la Universidad, o el escritorio de un miembro del grupo de trabajo, se convierte en un potencial nodo *worker* al contar con el software de base correspondiente. El conjunto de nueve estaciones de cálculo utilizadas para este trabajo, eventualmente en estado ocioso, se convierten en un conjunto de nueve estaciones receptoras de datos, cada una de ellas administrada por un objeto (hilo de ejecución) residente en el *master*. El *middleware* utilizado se ve materializado a través del conjunto de protocolos y servicios provistos por el *Globus Toolkit* [6], el cual tiene la funcionalidad necesaria para asegurar la autenticación de usuarios, envío y recepción de datos entre estaciones, etc. Este software de base provee un servicio denominado *gatekeeper* capaz de recibir la solicitud de ejecución de un trabajo desde un nodo *master* y realizar dicha tarea en forma local en el *worker*. Esto obliga a la instalación de este servicio en cada *worker*, con el objeto de satisfacer la ejecución del código de cálculo en lenguaje C. La infraestructura GRAM provista por el *Globus Toolkit* se utiliza para la localización y administración de recursos de procesamiento en una malla computacional. A través de GRAM la aplicación coordinadora, en este caso la aplicación Java residente en el *master*, solicita la ejecución de la tarea de cálculo residente en un nodo remoto, quedando el hilo responsable de esta tarea a la espera de la finalización de la misma. Cada "ejecución GRAM" es una sentencia más contenida dentro de las funciones de cada hilo. Esta tarea es llevada a cabo con la asistencia de *GridFtp*, aplicación provista también por el Globus Toolkit para la transferencia de ficheros entre nodos.

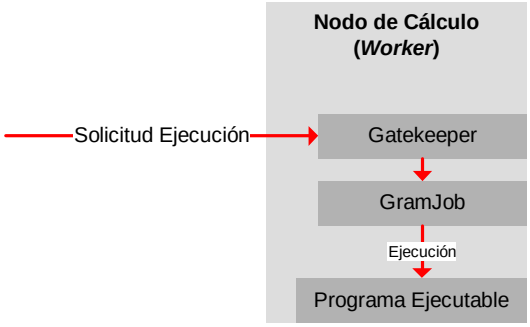

Figura. 3. La aplicación coordinadora invoca la ejecución sobre nodos remotos.

2.4. Implementación de los Hilos. Como se mencionó anteriormente, la solución final fué codificada en un programa Java, la cual tiene la posibilidad de manipular múltiples hilos de ejecución, donde cada uno de ellos representa un *objeto de datos* responsable de la administración de cada nodo *worker*. Al existir paralelismo entre los diferentes hilos de ejecución instanciados, se asegura el paralelismo de ejecución sobre los distintos recursos computacionales. Este paralelismo se obtiene a través del envío simultáneo de un mensaje especial a cada hilo (objeto *Thread*), "ordenándole" a los mismos el inicio de su ejecución. El código de cálculo residente en cada nodo *worker* complementa el sistema y es invocado por el servicio *gatekeeper* en forma local. Cada iteración sobre el conjunto de datos se transforma ahora en una invocación de ejecución; la que supone la ejecución del programa ejecutable en el nodo *worker* a través de los servicios GRAM. Esta dinámica dada por la ejecución del trabajo remoto y espera de los resultados procesados en cada iteración, se ve fortalecida por el manejo de hilos de ejecución de la aplicación coordinadora, permitiendo así definir tanto políticas de administración sobre cada uno de ellos, como así también observar tiempos de

ejecución sobre cada nodo entre otros datos. Previo a la invocación de la ejecución de tareas por parte de la aplicación coordinadora, esta debe descubrir qué recursos se encuentran actualmente disponibles, para lo cual hace uso de una función provista por una librería de funciones de Grid y aplicable en lenguaje java. Dicha función es la encargada de enviar paquetes de información hacia cada nodo *worker* quedando a la espera de la correspondiente respuesta. Cuando esta respuesta llega indica que el nodo *worker* está disponible para recibir solicitudes de ejecución.

Complementa este esquema una interfaz de usuario, plasmada en una aplicación WEB JSP [5], la cual tiene por objeto recolectar la información de entrada y características de ejecución provistas por el usuario final. De esta forma la aplicación *WEB* tiene acceso a la única credencial útil [1] para solicitar el uso de los recursos de la malla. Esta aplicación, se transforma ahora en la coordinadora de los distintos procesadores que calculan cada subconjunto de datos, pudiendo cada usuario final solicitar la ejecución de una instancia de la misma. Por último será esta aplicación la encargada de recolectar los resultados una vez finalizado el proceso iterativo, y presentar los mismos al usuario final. Este enfoque se ve enriquecido con el uso de portales *GRID*, donde se oculta al usuario la complejidad del uso de credenciales digitales, mejorando también la información sobre los recursos disponibles.

3. RESULTADOS OBTENIDOS

Se ha diseñado una aplicación capaz de emular el comportamiento paralelo para el cálculo de diferencias finitas a través de una malla computacional. Se observa que los tiempos se incrementan considerablemente debido a las características de la arquitectura utilizada: muy desacoplada y con una latencia muy alta. Sin embargo, la capacidad de escalabilidad de este modelo de ejecución no merece mayor comparación con la escalabilidad provista por un *cluster* de computadoras. La malla computacional con respecto al *cluster* presenta un escenario de mayor "libertad de crecimiento", como así también mayor incertidumbre de trabajo. La programación de aplicaciones en un ambiente de mallas computacionales puede presentar variados matices de acuerdo a la forma que toman los componentes de software y la forma en que son invocados.

Los autores estudian actualmente la migración de partes de la aplicación, las cuales se encuentran actualmente como archivos ejecutables, a *WebServices*. Por otro lado, se complementará el código Java con servicios provistos por Globus Toolkit para el seguimiento continuo de cada una de las tareas remotas.

4. FORMACION DE RECURSOS

Se trabaja en el departamento de Ingeniería en Sistemas de Información con alumnos del tercer año en temas relacionados con la programación concurrente, con el objeto de crear una relación directa con otros grupos de la Universidad dedicados al modelado numérico de aplicaciones; los cuales son potenciales usuarios de computación paralela.

5. BIBLIOGRAFIA

[1] J.Monetti & C.García Garino. Mallas Computacionales. Conceptos y Experiencias. 2º Congreso Internacional de Informática CIDI-Cuyo. Mendoza. 13-15 de abril de 2005.

[2] J. Monetti, C. Garcia Garino, O. León, Diseño y programación de aplicaciones para Grid Computing. Encuentro de Investigadores y Docentes de Ingeniería. ENIDI-2006. Mendoza, Argentina.

[3] MPI. www-unix.mcs.anl.gov/mpi.

[4] S. Chapra, C. Canale & P. Raymond, Métodos Numéricos para Ingenieros. Mc Graw-Hill. Cuarta Edición. 2004.

[5] H. Deitel & P.J. Deitel. Java. Compo Programar. Prentice Hall. Quinta Edición. 2004.

[6] The Globus Alliance. www.globus.org.