Ambiente distribuido de MPI en AWS para multiplicación de matrices

Maximiliano Alvarez & Daniel Camarena *ITAM*

28 de mayo de 2017

1. Introducción

A continuación presentaremos los detalles de la implementación de un programa que ejecuta la multiplicación de dos matrices (A, B) dentro de un cluster de servidores usando la librería *openMPI*.

1.1. Definición del problema

Dada una matriz A_{mxr}, con elementos a_{ij}, y una matriz B_{rxn} con elementos b_{ij}, la matriz C que resulta de multiplicar las matrices A y B es tal que cada uno de sus elementos i, j se calcula de la siguiente manera:

$$c_{ij} = \Sigma_{k=1}^{r} a_{ik} b_{kj} \tag{1}$$

Por cuestiones de simplicidad, este análisis y algoritmo se basa en matrices cuadradas de nxn.

Para cuantificar el trabajo asociado con el cálculo realizado por la máquina, consideraremos los flops asociados con la operación. Un flop es una operación de punto flotante: suma, multiplicación o división. El número de flops en un cómputo matricial típicamente se obtiene al sumar la cantidad de operaciones que se realizan.

Para la multiplicación de matrices mencionadas, el número de flops asociado a la operación es $2mnr$. Considerando únicamente el orden de magnitud de la operación, utilizamos la notación o grande, siendo esta operación de orden $O(n^3)$.

1.2. Objetivo

El objetivo de este proyecto es desarrollar un ambiente distribuido de MPI en Amazon Web Services (AWS) para realizar la operación de multiplicación de matrices de manera distribuida. Este ambiente distribuido será portátil, para poder realizar la multiplicación de matrices en cualquier instancia que nos encontremos.

1.3. Motivación

Dado que tenemos un algoritmo secuencial, es posible plantear este problema como uno a resolver de manera paralela, aprovechando múltiples procesadores para realizar el cómputo de la multiplicación de matrices.

1.4. Fundamentos

Un sistema de memoria distribuida consiste en una colección de pares core-memoria conectados por una red, y la memoria asociada a cada core solo puede ser accesada por ese core.

Existen diversas APIs (application programming interfaces) para la programación en los sistemas de memoria distribuida. Una de las APIs más populares es message passing interface, la cual es implementada de manera relativamente sencilla mediante la implementación de dos funciones principales: la función send (encargada de mandar tareas o datos del nodo maestro a los workers) y la función receive (encargada de recibir en el nodo maestro el resultado de los cálculos realizados por los workers).

Una implementación de esta API comúnmente utilizada para es openMPI, misma que fue utilizada durante la implementación de nuestro algorítmo de multiplicación de matrices.

MPI es una *message passing library interface specification*, dirigida al modelo de programación de message passing en paralelo. Entre sus objeticos se encuentran:

1. Ser un estándar para la escritura de message passing programs.
2. Diseño de una API.
3. Comunicación eficiente

MPI es una API poderosa y versátil, pero es de bajo nivel, en el sentido que hay una gran cantidad de detalle que el programador tiene que escribir. Las estructuras de datos tienen que ser replicadas por los procesos o ser explícitamente distribuidas.

2. Arquitectura

Dado el problema que buscamos resolver, necesitamos contar con un ambiente distribuído el cual puede ser construido con una serie de máquinas en la misma red o con servidores dentro de un servicio de *IaS* (Infrastructure as a Service). Para el caso partícular de este análisis escogimos contratar servidores dentro de Amazon Web Services debido a la facilidad de configuración y su bajo costo.

2.1. Amazon Web Services

Usamos AWS para crear un cluster basado en la nube. Esto nos provee una solución barata, flexible y escalable para implementar nuestro algoritmo. Nuestro cluster esta compuesto por servidores micro (ya que son sumamente económicos) con sistema operativo Ubuntu 14.04 los cuales serán preparados para poder ejecutar programas en paralelo, específicamente utilizando *openMPI*.

2.2. Arquitectura tipo de MPI en AWS

Para levantar un ambiente distribuido de MPI utilizando AWS, realizamos los siguientes pasos:

1. Levantamos N servidores, donde uno de ellos deberá de ser el nodo *master* y los demás *workers*.
2. Mediante la ayuda de *GNU Parallels* distribuiremos archivos alojados en nuestra máquina local a los servidores en AWS
3. Nos aseguramos que los servidores tengan instalados *GNU Parallels*
4. Con la ayuda de *Parallels* instalamos *openmpi 2.0.2* en todos nuestros servidores
5. Generamos un archivo de hosts con la ip publica y nombre de los servidores miembros del cluster y lo distribuimos a cada uno de los servidores
6. Usando la misma técnica, generamos un programa en C que haga la multiplicación de matrices en un cluster utilizando *openMPI*.
7. El archivo sourcefile configura algunas variables de ambiente en la consola en cada uno de los servidores
8. Generamos los archivos que contienen las matrices A y B y los distribuimos a cada uno de los servidores.

Todos estos pasos están contenidos en un archivo que orquesta la instalación del ambiente que necesitamos en cada uno de los servidores dentro de nuestro cluster. Para que esta instalación sea exitosa, requerimos que la máquina desde donde se hace la instalación cuente con el cliente de *AWS* y *GNU Parallels*.

2.3. Docker

Para poder hacer este ambiente portable y fácilmente reproducible utilizamos *Docker*, el cual nos permite generar máquinas virtuales que contengan las librerías de *openMPI* previamente instaladas y listas para ser utilizadas. La máquina virtual de *openMPI* fue publicada en *Docker Hub* con el siguiente tag: *johnydickens*

Una vez publicada esta maquina virtual en el *Docker Hub* podremos hacer uso de *Docker Swarm* para levantar un servicio distribuido

2.4. MPI Cluster en un Swarm de Docker-Compose

Usando *Docker-Compose*, es posible construir sistemas de múltiples partes, idóneos para el problema que estamos atendiendo. Sin embargo, estas partes corren de forma aislada. *Swarm* nos permite resolverlo, ya que es una herramienta para manejar un conjunto de *Docker* containers en un mismo host o en un cluster y conectarlos mediante el uso de una red virtual. Así, con *Swarm* podemos gestionar un cluster de *openMPI*.

Para poder armar el servicio de *Swarm* es necesario contar con un un archivo .yml en el cual se indica el nombre de cada uno de los contenedores, los puertos que cada contenedor tendrá expuesto. Además, en este

mismo archivo conectaremos los contenedores a una red virtual para asegurar que todos los contenedores se vean entre sí.

Si todo está bien ejecutado nuestra arquitectura portable estará formada por un nodo *maestro* y n *workers* (docker compose permite replicar un servicio n veces).

2.4.1. Complicaciones

EL cluster de *Swarm* es generado satisfactoriamente, se generan llaves ssh para que un servicio en un contenedor pueda hacer login en otro contenedor sin necesidad de usar una clave, sin embargo al momento de querer ejecutar un programa usando el *mpirun* o el *mpiexec* el cluster tarda un tiempo en responder y la respuesta que obtenemos es que no se pudo lograr la conexión entre los contenedores.

2.4.2. Pasos a seguir

Se hizo un debugging del cluster conectándonos a cada uno de los containers del servicio haciendo ping a los demas containers. La respuesta del ping fue exitosa en cada uno de los casos. El siguiente paso fue realizar una conexión ssh de un container a cada uno de los demás containers dentro del servicio, resultando en una resultado positivo en cada uno de los casos.

Valdrá la pena continuar realizando algunas pruebas y buscando documentación sobre puertos que deban estar abiertos dentro de las políticas de seguridad definidas para cada uno de los hosts que ejecutan a los containers.

3. Programa mpimm.c

Como se ha mencionado a lo largo de este artículo, se implementó un algoritmo de multiplicación de matrices distribuido utilizando la librería openMPI en C. Dicho algoritmo está implementado dentro del programa *mpimm.c*.

El programa *mpimm.c* consta de tres partes principales:

1. incialización y declaración de variables
2. inicialización de funciones propias de openMPI, específicamente las funciones send y recieve
3. distribuición de datos para paralelizar el cálculo de la matriz C = A * B

El programa *mpimm.c* está construido de tal forma que recibe las matrices A y B de archivos separados por coma (CSV), esto nos permite cambiar las matrices de entrada sin necesidad de recompilar el código. Además, las variables que alojan a cada una de las matrices A y B tienen alojamiento dinámico de memoria, lo cual nos da cierta flexibilidad sobre el tamaño de las matrices a multiplicar.

3.1. mpimm.c paso a paso

El programa mpimm.c consta de dos subrutinas y una rutina principal:

1. *readfile*: Recibe un apuntador a un archivo, un apuntador a un entero que indica el número de columnas y un apuntador a un entero que indica el número de renglones.
2. *allocate matrix*: Recibe el numero de renglones, numero de columnas y el archivo del cuál se leerá la información de la matriz, como resultado obtenemos la matriz en un formato pointer to pointer (**).

Vale la pena hacer la siguiente anotación: la subrutina *readfile* únicamente recorre el archivo indicado para saber cuantos renglones y cuantas columnas contiene algún archivo, posteriormente la subrutina *allocate matrix* será la encargada de cargar la matriz en memoria haciendo uso de *malloc*.

La rutina principal es la encargada de detonar las llamadas a cada una de estas subrutinas. Además distribuye datos entre los *workers* y recibe los cálculos parciales de cada uno de ellos para completar el cálculo de la multipliación de matrices. A continuación se presenta una explicación detallada de la rutina *main*

1. La rutina *main* comienza declarando las variables y apuntadores necesarios para el funcionamiento de nuestro algoritmo.
2. Se inicializa *MPI* y se define el *rank* y *size*. Además si no se cuenta con mas de dos tareas el programa no podrá ser ejecutado.
3. Se obtiene la información de la matriz A y B de los archivos correspondientes y se valida que la

multiplicación de estas dos matrices sea definida. (Estos tres pasos son ejecutados indistintamente en el nodo *maestro* y los *workers*)

4. Si el nodo es el maestro entonces se calculan los índices de los renglones que serán enviados a cada uno de los *workers*. En este caso cada *worker* calculará la multiplicación de la matriz usando bloques de renglones calculados de la siguiente manera:

$$rows = \frac{\#rows(A)}{\#workers} \tag{2}$$

   Para el caso particular en que $\#rows(A)\ \%\ \#workers \neq 0$, los renglones adicionales se envían al último de los *workers*.

5. Una vez que cada uno de los *workers* recibe la instrucción sobre qué rango de renglones de la matriz realizará la multiplicación, procede a calcular la entrada i, k de la siguiente forma:

$$c_{ik} = a_{ij} b_{jk} \tag{3}$$

   Para cada una de las entradas j de los i renglones de la matriz A que recibe.

6. Una vez calculadas todas las entradas de la matriz C, el nodo maestro la imprime en el *stdout*.

3.2. Compilación y Ejecución

Es importante subrayar que para que la ejecución en paralelo del programa *mpimm.c* funcione, este deberá ser compilado en cada uno de los nodos, para hacerlo se deberá ejecutar la siguiente instrucción en cada uno de ellos:

mpicc mpimm.c -o mpimm.out

Una vez compilado el programa en todos los nodos, se puede ejecutar desde el nodo maestro utilizando la siguiente instrucción:

mpirun –prefix /opt/openmpi-2.0.2/ -n 3 -H master,nodo1,nodo2 mpimm.out

4. Conclusiones

La implementación del programa *mpimm.c* representa un ejercicio de programación en paralelo utilizando *openMPI* y distribuyendo datos entre nuestros nodos. Durante este ejercicio pudimos comprobar que existe cierta complejidad al implementar la librería *openMPI* para crear programas en paralelo, sin embargo la curva de aprendizaje de esta librería es corta y existe documentación suficiente en la red para resolver los problemas que comúnmente se presentan durante la programación.

La implementación de la arquitectura también es compleja, se debe tener mucho cuidado y precisión al momento de crear la red entre los nodos y asegurar que las políticas de seguridad estén correctamente configuradas ya que de lo contrario la ejecución en paralelo no será exitosa.