
Ambiente distribuido de MPI en AWS para multiplicación de matrices

Maximiliano Alvarez & Daniel Camarena ITAM

30 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 $A_{m \times r}$, con elementos a_{ij} , y una matriz $B_{r \times n}$ 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:

$$c_{ij} = \sum_{k=1}^r a_{ik} b_{kj} \quad (1)$$

Por cuestiones de simplicidad, este análisis y algoritmo se basa en matrices cuadradas de $n \times n$.

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 algoritmo de multiplicación de matrices.

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

1. Ser un estándar para la escritura de message passing programs.
2. Diseño de una API.
3. Comunicación eficiente
3. Con la ayuda de *GNU Parallels*, instalamos *openmpi 2.0.2* en todos nuestros servidores
4. Generamos un archivo de hosts con la ip publica y nombre de los servidores miembros del cluster y lo distribuimos a cada uno de estos servidores.

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 distribuido, el cual puede ser construido a partir de una serie de máquinas en una misma red o con servidores dentro de un servicio Infrastructure as a Service (*IaaS*). Para el caso particular de este análisis, optamos por contratar servidores dentro de Amazon Web Services (AWS), 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 nodos serán *workers*.
2. Mediante la aplicación de *GNU Parallels*, distribuiremos archivos alojados en nuestra máquina local a los servidores en AWS (para ello, es necesario asegurarnos que este esté instalado en todos los servidores).

5. Usando el mismo procedimiento, generamos un programa en C que haga la multiplicación de matrices en un cluster utilizando *openMPI*.
6. Usando el archivo sourcefile, configuramos algunas variables de ambiente en la consola en cada uno de los servidores.
7. 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: *johnnydickens*).

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, cada una de 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 construir el servicio de Swarm, es necesario contar con un archivo `.yaml`, en el cual se indique el nombre de cada uno de los contenedores y 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 *master* y *n workers* (Docker Compose permite replicar un servicio *n* veces).

2.4.1. Complicaciones

El cluster de Swarm es generado satisfactoriamente y 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 *mpirun* o *mpiexec*, el cluster tarda un tiempo significativo en responder y la respuesta que obtenemos indica 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 demás 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 un resultado positivo en cada uno de los casos.

Valdría 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. Algoritmo

3.1. 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. Inicialización y declaración de variables.
2. Inicialización de funciones propias de *openMPI*, específicamente las funciones *send* y *receive*.
3. Distribución 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.

3.2. *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 número de renglones, número de columnas y el archivo del cual se leerá la información de la matriz; como resultado obtenemos la matriz en un formato *pointer to pointer* (**).

Nota a considerar: la subrutina *readfile* únicamente recorre el archivo indicado para saber cuántos renglones y cuántas 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 multiplicación 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 del algoritmo.

2. Se inicializa MPI y se define el *rank* y *size* (si no se cuenta con más 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 *master* y los nodos *workers*).
4. Si el nodo es el *master*, 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 correspondientes, obtenidos de la siguiente manera:

$$rows = \frac{number\ of\ rows(A)}{number\ of\ workers} \quad (2)$$

Para el caso particular en que $number\ of\ rows(A) \% number\ of\ workers \neq 0$, 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:

$$c_{ik} = a_{ij}b_{jk} \quad (3)$$

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 *master* la imprime en el *stdout*.

3.3. 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:

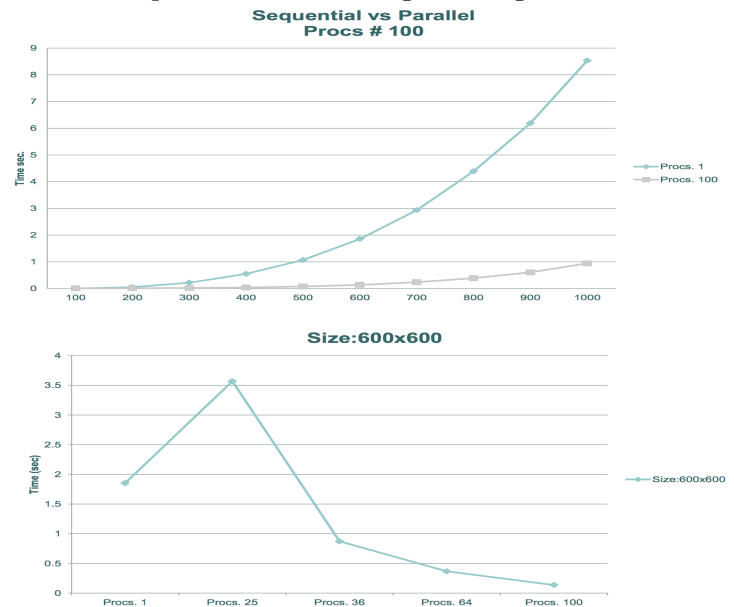
- `mpicc mpimm.c -o mpimm.out`

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

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

4. Resultados

Como resultados de este ejercicio obtuvimos un cluster de servidores Ubuntu 14.04 con la librería de openMP completamente instalada y funcionando. Se ejecutaron varias pruebas corriendo el programa para matrices cuadradas y matrices rectangulares teniendo un performance similar al ejecutar en modo distribuido y semi distribuido. Por razones de falta de tiempo no fue posible hacer comparaciones de tiempo de ejecución entre una versión secuencial de este algoritmo y la versión en paralelo, sin embargo el performance esperado es el presentado en las siguientes gráficas:



El hecho de contar con un servicio como el de AWS nos dió las siguientes ventajas:

1. Dado que no estamos sujetos a las limitaciones del hardware de nuestra máquina local, es posible escalar indefinidamente este cluster.
2. Es una alternativa barata, dados los costos del uso de máquinas de AWS.

A pesar de que cada uno de los nodos *workers* recibe a las matrices *A* y *B* completas, los cálculos asociados con la multiplicación de matrices se distribuyen de manera eficiente. Esto implica que, para matrices grandes, es posible observar mejoras en el desempeño de la multiplicación de matrices con nuestro algoritmo sobre una ejecución secuencial del mismo proceso.

5. 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 de prestar especial atención y cuidado durante la creación de 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.