UNIVERSIDAD DE COLIMA

Facultad de Telemática

Ingeniería en Software

Práctica #11
“Algoritmo paralelo de búsqueda binaria”

Ramírez García Gilberto Felipe

Valencia Sandoval Alfonso

Velasco Munguía Roberto Antonio

5 - “K”

Prof. Ramirez Alcaraz Juan Manuel

29 de Noviembre del 2019

Índice

Introducción

Algoritmo paralelo de búsqueda binaria

Conexión con el cluster

El primer paso para realizar esta práctica será establecer conexión al cluster, tal como lo hemos visto en prácticas anteriores.

Para mantener orden en la práctica, creamos un directorio de trabajo para almacenar todos nuestros archivos y códigos de programación de este programa.

El algoritmo secuencial y el paralelo

Algoritmo secuencial

El algoritmo secuencial de búsqueda binaria es un algoritmo para encontrar un elemento en una lista ordenada de elementos. El algoritmo consiste en dividir a la mitad la lista y tomar el elemento de en medio como nodo raíz. Si el target es menor a la raíz, entonces se repite la operación de división con los elementos comprendidos entre el nodo raíz y el inicio de la lista. Por el contrario, si el target es mayor que el nodo raíz, entonces se repite la operación de división, pero ahora con los elementos comprendidos entre el nodo raíz y el final de la lista. Así hasta reducir las ubicaciones posibles a una sola. El proceso termina cuando el nodo raíz coincide con el target. Como se puede apreciar, este algoritmo va descartando la mitad de los elementos en cada iteración. Por eso, es un algoritmo eficiente. Su tiempo logarítmico es O (log n).

Algoritmo paralelo

El algoritmo secuencial lo paralelizamos dividiendo los datos y asignando cada uno a un proceso. Es decir, dividimos los elementos de la lista en k grupos; k debe de ser menor al número de procesadores. N es el número total de elementos. Así pues, dividimos la lista en n / k grupos y con la función MPI_Scatter asignamos un grupo a cada proceso. Cada proceso ejecuta la búsqueda binaria en su grupo de datos correspondiente. El proceso que encuentra el target lo imprime en pantalla. Su tiempo logarítmico es O log (n/k).

Codificación del programa

Escribimos nuestro programa en un archivo de texto con extensión "*.c*". En nuestro caso utilizamos un editor de código local para poder hacer uso de herramientas de corrección y resaltado de código, por lo que posteriormente subiremos el archivo al cluster utilizando *scp*.

Explicación del código

```
void fill_array(int n, int arr[n]){
 for(int i = 0; i < n; i++)  {
   arr[i] = i + 1;
 }
}
```

Primero tenemos una función llamada fill_array el cual se encarga de recibir una referencia de un arreglo de tamaño n, e inicializarlo con números secuenciales definidos como *i + 1*. Donde *i* representa el índice de cada elemento del arreglo.

```
void binary_search (int length, int array[length], int target, int
proc_id){

 int start = 0;
 int end = length - 1;

 if (target > array[start] && target < array[end]) {
   int search_result = 0;

   while(start <= end && array[search_result] != target ) {

     search_result = (start + end) / 2;

     if (array[search_result] == target)
       printf("Target %d found by process %d!\n", target, proc_id);
     else if (array[search_result] < target)
       start = search_result + 1;
     else
       end = search_result -1;
   }
 } else
   printf("Target %d couldn't be found by process %d\n", target,
           proc_id);
}
```

Seguido tenemos una función llamada binary_search. Donde dado cualquier arreglo de números enteros ordenados de menor a mayor, y un nṁero objetivo a buscar, somete el arreglo a un algoritmo de búsqueda binaria. Como cada proceso ejecutará un bloque del arreglo lo primero que hacemos es comparar si el número que buscamos está entre el rango de valores del arreglo, y así valoramos si es necesario ejecutar el algoritmo o no.

El primer paso del algoritmo es conocer el inicio y el final del rango de valores donde se puede encontrar nuestro objetivo; por lo que tenemos la variable *start* igual al primer índice del arreglo, y la variable *end* igualada al último índice del arreglo.

Luego, encontramos el índice medio entre *start* y *end*. Comparamos si el valor en este índice es igual al valor que buscamos, de lo contrario si el valor medio tomado es mayor que el que buscamos, entonces repetimos el proceso en la primer mitad de este rango de valores, igualando la variable *end* al índice medio menos 1. De lo contrario si el valor medio tomado es menor que el número que buscamos, entonces repetimos el proceso en la segunda mitad de este rango valores, igualando la variable *start* al índice medio más 1. Este es básicamente el algoritmo de búsqueda binaria, el cual en un punto encontrará el valor objetivo.

```
int proc_id, total_procs;

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &total_procs);
MPI_Comm_rank(MPI_COMM_WORLD, &proc_id);

...

MPI_Finalize();
```

El siguiente punto en el programa es la ejecución principal, donde primero inicializamos MPI con el método *MPI_Init*, conseguimos el total de procesos asignado al trabajo con *MPI_Comm_Size* y conseguimos el índice del proceso actual con *MPI_Comm_Rank*. Después de la ejecución de nuestro programa finalizamos MPI utilizando *MPI_Finalize*.

```
if (argc > 1) {
   int collection_length = atoi(argv[1]);
   int target = atoi(argv[2]);

   ...


} else if (proc_id == 0)
   printf("Program usage: binary-search <collection_length>
            <target>\n");
```

Para definir la cantidad de elementos que tendrá nuestro arreglo de datos en el que aplicaremos la búsqueda y definir qué número buscar, decidimos utilizar argumentos. Por lo que limitamos la ejecución del programa a que se hayan ingresado el número correcto de argumentos y los almacenamos en las variables *collection_length* y *target*. Si no se proporcionan los argumentos necesarios, el programa no se ejecutará y solo imprimirá la forma correcta de utilizarse.

```
if (collection_length % total_procs == 0 && total_procs <
```

```
collection_length) {
} else if (proc_id == 0)
     printf("Can't execute program: More processes than collection
            elements or there is no precise block size!\n");
```

Para evitar problemas con el tamaño del bloque, condicionamos la ejecución del programa que el tamaño del arreglo debe ser múltiplo del total de procesos y que no existan más procesos que elementos dentro del arreglo.

```
int data[collection_length];
int block_size = collection_length / total_procs;
int block[block_size];

if (proc_id == 0)
  fill_array(collection_length, data);

MPI_Scatter(data, block_size, MPI_INT, block, block_size, MPI_INT, 0,
            MPI_COMM_WORLD);
binary_search(block_size, block, target, proc_id);
```

Finalmente, tenemos el corazón del programa, donde el proceso 0 se encargará de inicializar el array de tamaño n definido por *data[collection_length]*. Calculamos el tamaño de cada bloque de datos que recibirá cada proceso con *block_size = collection_length / total_procs* y creamos la variable *block[block_size]* donde cada proceso recibiŕa su respectivo bloque.

Siguiente, utilizamos *MPI_Scatter* para repartir los bloques a cada proceso, definiendo *data* como el arreglo de datos que será transmitido, *block_size* como la cantidad de elementos que enviará el proceso 0 a cada otro proceso, el tipo de dato que se envía en este caso números enteros definido con *MPI_INT*, block como la variable en donde se recibirán cada bloque de datos, *block_size* como la cantidad de elementos que se recibirán, el tipo de dato que se recibe en este caso números enteros definido con *MPI_INT*, el índice del proceso que enviará la información en este caso el proceso 0; por último el comunicador al cual se enviarán los datos en este caso *MPI_COMM_WORLD*.

Y por último, cada proceso ejecutará la busqueda binaria en el respectivo bloque de datos que se le asignó.

Desplegando nuestro trabajo en el cluster

Cargamos los módulos y compilamos el programa binary_search.c

Adaptamos el scritp mpi-run.sh para este ejercicio:

```
[curso13@rogueone binary-search]$ module load Compilers/Parallel-Studio-XE-2018
[curso13@rogueone binary-search]$ ls
binary.log  binary-search    machines.LINUX
binary.out  binary-search.c  mpi-run.sh
[curso13@rogueone binary-search]$ mpiicc binary-search.c -o binary-search
binary-search.c(24): warning #266: function "fill_array" declared implicitly
          fill_array(collection_length, data);
          ^

binary-search.c(27): warning #266: function "binary_search" declared implicitly
        binary_search(block_size, block, target, proc_id);
        ^

binary-search.c(39): warning #159: declaration is incompatible with previous "fi
ll_array" (declared at line 24)
  void fill_array(int n, int arr[n]){
       ^

binary-search.c(45): warning #159: declaration is incompatible with previous "bi
nary_search" (declared at line 27)
  void binary_search (int length, int array[length], int target, int proc_id){
       ^
```

```
#!/bin/bash

#PBS -l nodes=2:ppn=2,walltime=00:00:10
#PBS -N mpi_binary_search
#PBS -q staff
#PBS -d /mnt/zfs-pool/home/curso13/Poncho/binary-search/
#PBS -o binary.log
#PBS -j oe
#PBS -V
#PBS -S /bin/bash

source $MODULESHOME/init/bash
module purge
module load Compilers/Parallel-Studio-XE-2018

NPROCS=`wc -l < $PBS_NODEFILE`
cat ${PBS_NODEFILE} | sort -u > $PBS_O_WORKDIR/machines.LINUX

mpirun -np $NPROCS -machinefile machines.LINUX ./binary-search 4000 3257 > binar
y.out
```

Ejecución del programa

Enviamos el script a la cola de ejecución, para seguir las políticas del clúster.

```
[curso13@rogueone binary-search]$ qsub mpi-run.sh
8017.rogueone
```

Revisamos su estatus antes de terminar ejecución:

```
8017.rogueone              ...binary_search curso13                0 R staff
```

Después de terminar ejecución:

```
8017.rogueone              ...binary_search curso13        00:00:00 C staff
```

Captura del resultado

Capturamos los mensajes de consola y observamos que el programa se ha ejecutado correctamente.

En este ejemplo se utilizaron 2 nodos con 2 procesadores cada uno, esto es un total de 4 procesadores. Se buscó el número 3257 dentro de un vector de 4000 elementos. Como podemos observar en el resultado, se dividió en 4 bloques y cada bloque fue asignado a un procesador, el 3er procesador encontró el número objetivo.

```
Target 3257 couldn't be found by process 2
Target 3257 couldn't be found by process 0
Target 3257 couldn't be found by process 1
Target 3257 found by process 3!
```

En este ejemplo se utilizaron 4 nodos con 2 procesadores cada uno, esto es un total de 8 procesadores. Se buscó el número 3257 dentro de una vector de 4000 elementos. Como podemos observar en el resultado, se dividió en 8 bloques y cada bloque fue asignado a un procesador, el 6to procesador encontró el número objetivo.

```
[curso13@rogueone binary-search]$ cat binary.out
Target 3257 couldn't be found by process 4
Target 3257 found by process 6!
Target 3257 couldn't be found by process 2
Target 3257 couldn't be found by process 5
Target 3257 couldn't be found by process 1
Target 3257 couldn't be found by process 7
Target 3257 couldn't be found by process 3
Target 3257 couldn't be found by process 0
```

En este ejemplo se utilizaron 8 nodos con 2 procesadores cada uno, esto es un total de 16 procesadores. Se buscó el número 3257 dentro de una vector de 4000 elementos. Como podemos observar en el resultado, se dividió en 16 bloques y cada bloque fue asignado a un procesador, el 13er procesador encontró el número objetivo.

```
[curso13@rogueone binary-search]$ cat binary.out
Target 3257 couldn't be found by process 8
Target 3257 couldn't be found by process 2
Target 3257 couldn't be found by process 10
Target 3257 couldn't be found by process 4
Target 3257 couldn't be found by process 12
Target 3257 couldn't be found by process 14
Target 3257 couldn't be found by process 6
Target 3257 couldn't be found by process 0
Target 3257 couldn't be found by process 3
Target 3257 couldn't be found by process 1
Target 3257 couldn't be found by process 11
Target 3257 couldn't be found by process 9
Target 3257 couldn't be found by process 15
Target 3257 couldn't be found by process 7
Target 3257 found by process 13!
Target 3257 couldn't be found by process 5
```

En este ejemplo se utilizaron 4 nodos con 2 procesadores cada uno, esto es un total de 4 procesadores. Se buscó el número 3257 dentro de un vector de 5000 elementos. Como podemos observar en el resultado se dividió en 4 bloques y cada bloque fue asignado a un procesador, el 5to procesador encontró el número objetivo.

```
Target 3257 couldn't be found by process 4
Target 3257 found by process 5!
Target 3257 couldn't be found by process 1
Target 3257 couldn't be found by process 6
Target 3257 couldn't be found by process 2
Target 3257 couldn't be found by process 3
Target 3257 couldn't be found by process 7
Target 3257 couldn't be found by process 0
```

El tamaño del vector fue cambiado a 8000 elementos y se mantuvieron los datos mencionados anteriormente.

```
Target 3257 couldn't be found by process 5
Target 3257 couldn't be found by process 1
Target 3257 couldn't be found by process 4
Target 3257 couldn't be found by process 6
Target 3257 found by process 3!
Target 3257 couldn't be found by process 2
Target 3257 couldn't be found by process 7
Target 3257 couldn't be found by process 0
```

Y por último el tamaño del vector fue de 12000 elementos, con los datos antes mencionados.

```
Target 3257 couldn't be found by process 3
Target 3257 couldn't be found by process 7
Target 3257 couldn't be found by process 1
Target 3257 couldn't be found by process 4
Target 3257 couldn't be found by process 5
Target 3257 found by process 2!
Target 3257 couldn't be found by process 6
Target 3257 couldn't be found by process 0
```

Conclusión

Observamos que primero tienes que comprender el algoritmo secuencial. Ya que lo entiendes, ves cómo puedes paralelizar este. Esto lo comentamos, porque tuvimos un poco de dificultad en entender el secuencial.

En conclusión, el algoritmo secuencial de búsqueda binaria es rápido; pero el algoritmo paralelo de búsqueda binaria es todavía más rápido. Esto gracias a que puedes dividir los datos y ejecutar el algoritmo secuencial en muchos procesadores al mismo tiempo.

Referencias

Khan Academy, Búsqueda binaria. (2019). Recuperado de http://bit.ly/2Lj4xMH

Kumbhar, V., (2019). Parallel Search Algorithms using MPI. Recuperado de http://bit.ly/2QZ3lBC

Wikipedia. Búsqueda binaria. (s.f.). Recuperado de http://bit.ly/35NPFh3

Código del algoritmo paralelo

```
#include<stdio.h>
#include <stdlib.h>
#include<mpi.h>

int main(int argc, char* argv[]) {

 int proc_id, total_procs;

 MPI_Init(&argc, &argv);
 MPI_Comm_size(MPI_COMM_WORLD, &total_procs);
 MPI_Comm_rank(MPI_COMM_WORLD, &proc_id);

 if (argc > 1) {
   int collection_length = atoi(argv[1]);
   int target = atoi(argv[2]);

      if (collection_length % total_procs == 0 && total_procs < collection_length) {

     int data[collection_length];
     int block_size = collection_length / total_procs;
     int block[block_size];

     if (proc_id == 0)
       fill_array(collection_length, data);
```

```
     MPI_Scatter(data, block_size, MPI_INT, block, block_size, MPI_INT,
0, MPI_COMM_WORLD);
     binary_search(block_size, block, target, proc_id);
   } else if (proc_id == 0)
       printf("Can't execute program: More processes than collection
elements or there is no precise block size!\n");
 } else if (proc_id == 0)
        printf("Program usage: binary-search <collection_length>
<target>\n");

 MPI_Finalize();

 return 0;

}

void fill_array(int n, int arr[n]){
 for(int i = 0; i < n; i++)  {
   arr[i] = i + 1;
 }
}

void binary_search (int length, int array[length], int target, int
proc_id){

 int start = 0;
 int end = length - 1;

 if (target > array[start] && target < array[end]) {
   int search_result = 0;

   while(start <= end && array[search_result] != target ) {

     search_result = (start + end) / 2;

     if (array[search_result] == target)
       printf("Target %d found by process %d!\n", target, proc_id);
     else if (array[search_result] < target)
       start = search_result + 1;
     else
       end = search_result -1;
   }
 } else
     printf("Target %d couldn't be found by process %d\n", target,
proc_id);
}
```