Instituto Politécnico Nacional.

Escuela Superior De Cómputo.

Materia:

Aplicaciones Para Comunicaciones En Red.

Tema:

Reporte.

(Práctica 5).

Profesor:

Axel Ernesto Moreno Cervantes.

Alumnos:

Luis Enrique Rojas Alvarado.

Mario Alberto Miranda Sandoval.

Grupo:

3CM5.

Introducción.

Sort Bucket.

La clasificación de cubetas, o clasificación de cubetas, es un algoritmo de clasificación que funciona distribuyendo los elementos de una matriz en una serie de cubetas. Cada cubo se clasifica individualmente, ya sea utilizando un algoritmo de clasificación diferente o aplicando recursivamente el algoritmo de clasificación de cubo.

Es un tipo de distribución, una generalización del tipo de casillero, y es un primo de tipo radix. La clasificación de cubetas se puede implementar con comparaciones y, por lo tanto, también se puede considerar un algoritmo de clasificación de comparación.

La complejidad computacional depende del algoritmo utilizado para clasificar cada depósito, el número de depósitos que se utilizarán y si la entrada se distribuye uniformemente.

La clasificación de cubetas funciona de la siguiente manera:

- Configure una matriz de "cubos" inicialmente vacíos.
- Dispersión: vaya sobre la matriz original, colocando cada objeto en su cubo.
- Ordenar cada cubo no vacío.
- Recopilar: visite los cubos en orden y vuelva a colocar todos los elementos en la matriz original.

Merge Sort.

Algoritmo basado en la técnica de diseño de algoritmos Divide y Vencerás.

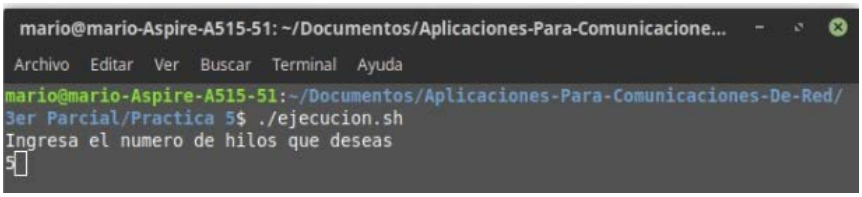
Ordenar una secuencia S de elementos:

- Si S tiene uno o ningún elemento, está ordenada
- Si S tiene al menos dos elementos se divide en dos secuencias S1 y S2, S1 conteniendo los primeros n/2, y S2 los restantes.
- Ordenar S1 y S2, aplicando recursivamente este procedimiento.
- Mezclar S1 y S2 ordenadamente en S
- Mezclar dos secuencias ordenadas S1 y S2 en S:
- Se tienen referencias al principio de cada una de las secuencias a mezclar (S1 y S2).

- Mientras en alguna secuencia queden elementos, se inserta en la secuencia resultante (S) el menor de los elementos referenciados y se avanza esa referencia una posición.

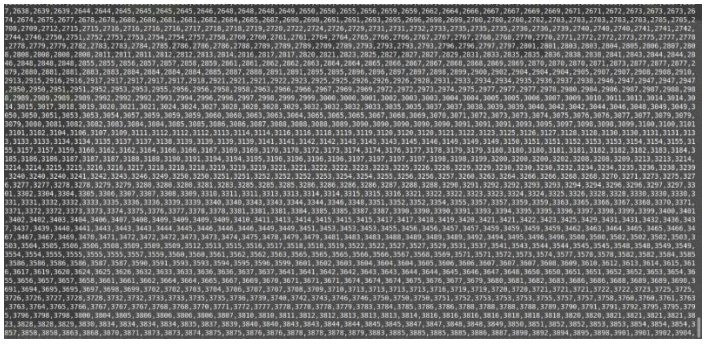
Desarrollo.

Para el desarrollo de esta práctica se implemento en lenguaje C, debido a que es más sencillo manejar los descriptores de los sockets.



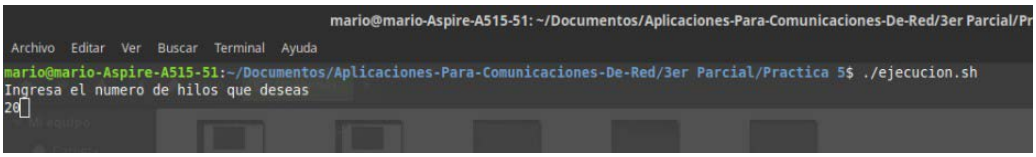
Primero se solicita al usuario que introduzca un número de hilos que desea, ya que este número obedece a: $2 * n$, donde es el numero total de hilos dividido entre servidores y clientes, además de indicar el número de clientes.

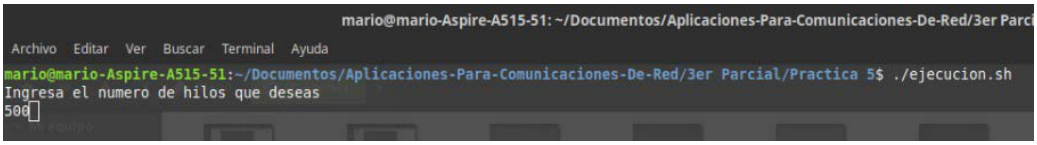
Se crean y levantan los servidores, y se muestra su estado de conexión con los clientes.



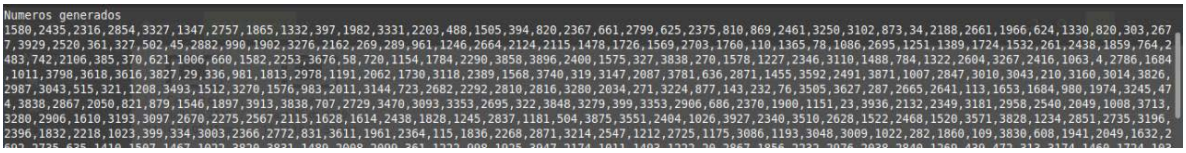
Se muestran los números ordenados.

Pruebas.



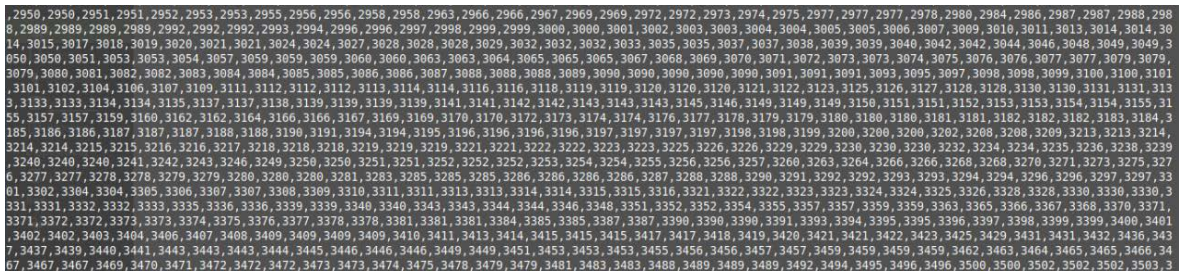


Petición de la cantidad de hilos.



Impresión de los números generados.

Creación de los servidores.



Números ordenados.

Código.

Sockets.h

```h
#ifndef __SOCKETS__
#define __SOCKETS__

#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <netdb.h>

int levantarServidor(const char *);
int levantarCliente(const char *);

#endif //__SOCKETS__
```

Sockets.c

```c
#include "sockets.h"

int levantarServidor(const char *puerto) {
    struct addrinfo *servinfo, *p;
    struct addrinfo hints;
    int rv, sd, v = 1;

    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_INET6;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE;
    hints.ai_protocol = 0;
    hints.ai_canonname = NULL;
    hints.ai_addr = NULL;
    hints.ai_next = NULL;

    if((rv = getaddrinfo("localhost", puerto, &hints, &servinfo)) != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
        return 1;
    }

    for(p = servinfo; p != NULL; p = p -> ai_next) {
        if((sd = socket(p -> ai_family, p -> ai_socktype, p -> ai_protocol)) == -1) {
            perror("server: socket");
            continue;
        }

        if(setsockopt(sd, SOL_SOCKET, SO_REUSEADDR, &v, sizeof(int)) == -1) {
            perror("setsockopt");
            close(sd);
            exit(EXIT_FAILURE);
        }

        if(bind(sd, p -> ai_addr, p -> ai_addrlen) == -1) {
            close(sd);
            perror("server: bind");
            continue;
```

```
38.         }
39.         break;
40.     }
41.
42.     freeaddrinfo(servinfo);
43.
44.     if(p == NULL) {
45.         fprintf(stderr, "servidor: error en bind\n");
46.         exit(1);
47.     }
48.
49.     if(listen(sd, 5) == -1) {
50.         perror("server: listen");
51.         exit(1);
52.     }
53.
54.     return sd;
55. }
56.
57. int levantarCliente(const char *puerto) {
58.     struct addrinfo *cliente, *p;
59.     struct addrinfo hints;
60.     int rv, cd, v = 1;
61.     char *srv = "2001::1234:0001";
62.
63.     memset(&hints, 0, sizeof(hints));
64.     hints.ai_family = AF_INET6;
65.     hints.ai_socktype = SOCK_STREAM;
66.     hints.ai_protocol = 0;
67.
68.     if((rv = getaddrinfo("localhost", puerto, &hints, &cliente)) != 0) {
69.         fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
70.         return 1;
71.     }
72.
73.     for(p = cliente; p != NULL; p = p -> ai_next) {
74.         if((cd = socket(p -> ai_family, p -> ai_socktype, p -> ai_protocol)) == -
    1) {
75.             perror("client: socket");
76.             continue;
77.         }
78.
79.         if(connect(cd, p -> ai_addr, p -> ai_addrlen) == -1) {
80.                 close(cd);
81.                 perror("client: connect");
82.                 continue;
83.         }
84.
85.         break;
86.     }
87.
88.     if(p == NULL) {
89.         fprintf(stderr, "client: error al conectar con el servidor\n");
90.         return 2;
91.     }
92.
93.     freeaddrinfo(cliente);
94.
95.     return cd;
96. }
```

Cubetas.h

```
#ifndef __CUBETAS__
#define __CUBETAS__

#include <time.h>
#include <pthread.h>
#include "sockets.h"
#include "MergeSort.h"

struct dato {
    int tam;
    int *numeros;
};

void *cliente(void *);
void *servidor(void *);
void crearNumeros();
void crearClientes(int);
void crearServidores(int);
int ingresar(int *, int);

int numeros[4000];
int final[4000];

#endif // !__CUBETAS__
```

Cubetas.c

```
#include "cubetas.h"

void crearServidores(int numeroServidores) {
    pthread_t hilos[numeroServidores];

    for(int i = 0; i < numeroServidores; i += 1) {
        int *j = malloc(sizeof(int));
        *j = i;
        int e = pthread_create(&hilos[i], NULL, &servidor, (void *)j);

        if(e)
            exit(EXIT_FAILURE);
    }

    sleep(1);
}

void crearClientes(int numeroClientes) {
    int rango = 4000/numeroClientes;
    pthread_t hilos[numeroClientes];
    int *indices;
    int inicio = 0;
    int error;

    for(int i = 0; i < numeroClientes; ++i) {
        indices = malloc(sizeof(int) * 3);
        indices[0] = i;
        indices[1] = inicio;
        inicio += rango;
        indices[2] = inicio;
```

```
31.
32.         if(i == numeroClientes - 1 && inicio < 4000)
33.             indices[2] = 4000;
34.
35.         int e = pthread_create(&hilos[i], NULL, cliente, (void *)indices);
36.
37.         if(e)
38.             exit(EXIT_FAILURE);
39.     }
40.
41.     int iniciar = 0;
42.     for(int i = 0; i < numeroClientes; i += 1) {
43.         int *ordenados;
44.
45.         if(pthread_join(hilos[i], (void **) &ordenados) != 0)
46.             exit(EXIT_FAILURE);
47.
48.         iniciar = ingresar(ordenados, iniciar);
49.     }
50. }
51.
52. int ingresar(int *ordenados, int inicio) {
53.     int i = 0;
54.
55.     while(ordenados[i] != -1) {
56.         final[inicio++] = ordenados[i++];
57.     }
58.
59.     return inicio;
60. }
61.
62. void crearNumeros() {
63.     srand(time(NULL));
64.
65.     for(int i = 0; i < 4000; i += 1) {
66.         numeros[i] = rand() % 4000;
67.         printf("%d,", numeros[i]);
68.     }
69.
70.     printf("\n");
71. }
72.
73. void *servidor(void *indice) {
74.     int *datitos = (int *) indice;
75.     int puerto = 8000 + datitos[0];
76.     char pto[10];
77.     snprintf(pto, 10, "%d", puerto);
78.
79.     int socketServidor = levantarServidor(pto);
80.     printf("Servidor #%d listo\n", datitos[0]);
81.     int socket;
82.
83.     socklen_t ctam;
84.     char hbuf[NI_MAXHOST];
85.     char sbuf[NI_MAXSERV];
86.
87.     struct sockaddr_storage c_addr;
88.
89.     for(;;) {
90.         ctam = sizeof(c_addr);
91.         socket = accept(socketServidor, (struct sockaddr *)&c_addr, &ctam);
```

```
92.
93.         if(socket == -1) {
94.             perror("Server: accept");
95.             continue;
96.         }
97.
98.         if(getnameinfo((struct sockaddr *)&c_addr, sizeof(c_addr), hbuf, sizeof(hb
    uf), sbuf, sizeof(sbuf), NI_NUMERICHOST | NI_NUMERICSERV) == 0)
99.             printf("Cliente conectado desde %s:%s\n", hbuf, sbuf);
100.
101.                char buffer[4000];
102.                memset(&buffer, 0, sizeof(buffer));
103.                read(socket, buffer, sizeof(buffer));
104.                struct dato *datos = (struct dato *) buffer;
105.                ordenamientoMerge(datos -> numeros, 0, datos -> tam - 1);
106.                write(socket, (const char *)datos, sizeof(struct dato));
107.                close(socket);
108.
109.                break;
110.            }
111.
112.            close(socketServidor);
113.            return NULL;
114.        }
115.
116.        void *cliente(void *indices) {
117.            int *datitos = (int *) indices;
118.            int desordenados[4000];
119.            int j = 0;
120.
121.            for(int i = 0; i < 4000; i += 1) {
122.                if(numeros[i] >= datitos[1] && numeros[i] < datitos[2]) {
123.                    desordenados[j++] = numeros[i];
124.                }
125.            }
126.
127.            int puerto = 8000 + datitos[0];
128.            char pto[10];
129.            snprintf(pto, 10, "%d", puerto);
130.            int socket = levantarCliente(pto);
131.
132.            struct dato *datos = malloc(sizeof(struct dato));
133.            datos -> tam = j;
134.            datos -> numeros = desordenados;
135.            write(socket, (const char *) datos, sizeof(struct dato));
136.
137.            char buffer[4000];
138.            memset(&buffer, 0, sizeof(buffer));
139.            read(socket, buffer, sizeof(buffer));
140.
141.            struct dato *datosDevueltos = (struct dato *) buffer;
142.            int *ordenados = malloc((j + 1) * sizeof(int));
143.
144.            for(int i = 0; i < j; i += 1)
145.                ordenados[i] = datosDevueltos -> numeros[i];
146.
147.            ordenados[j] = -1;
148.            close(socket);
149.            return (void *) ordenados;
150.        }
```

MergeSort.h

```h
#ifndef __MERGESORT__
#define __MERGESORT__

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

void mezclaMerge(int* numeros, int bajo, int mitad, int alto);
void ordenamientoMerge(int* numeros, int bajo, int alto);

#endif // !__MERGESORT__
```

MergeSort.c

```c
#include "MergeSort.h"

void ordenamientoMerge(int* numeros, int bajo, int alto){
    if(bajo<alto){
        int mitad=bajo+(alto-bajo)/2;
        ordenamientoMerge(numeros, bajo, mitad);
        ordenamientoMerge(numeros, mitad+1, alto);
        mezclaMerge(numeros, bajo, mitad, alto);
    }
}

void mezclaMerge(int* numeros, int bajo, int mitad, int alto){
    int dimension1=mitad-bajo+1;
    int dimension2=alto-mitad;
    int i,j,k;
    int A[dimension1], B[dimension2];

    for(i=0; i<dimension1; i++)
        A[i]=numeros[bajo+i];
    for(j=0; j<dimension2; j++)
        B[j]=numeros[mitad+1+j];

    i=0;
    j=0;
    k=bajo;

    while(i<dimension1 && j<dimension2){
        if(A[i]<=B[j]){
            numeros[k]=A[i];
            i++;
        }else{
            numeros[k]=B[j];
            j++;
        }
        k++;
    }

    while(i<dimension1){
        numeros[k]=A[i];
        k++;
        i++;
    }

    while(j<dimension2){
```

```
        numeros[k]=B[j];
        k++;
        j++;
    }

}
```

Principal.c

```
#include "cubetas.h"

int main() {
    int chilos = 0;

    printf("Ingresa el numero de hilos que deseas\n");
    scanf("%d", &chilos);

    printf("Numeros generados\n");
    crearNumeros();

    crearServidores(chilos);
    crearClientes(chilos);

    printf("Numeros Ordenados\n");

    for(int i = 0; i < 4000; i += 1) {
        printf("%d,", final[i]);
    }

    printf("\n");
    return 0;
}
```

Conclusión.

Miranda Sandoval Mario Alberto.

En esta práctica se pudo observar como un algoritmo como el bucket sort puede ser implementado en red, donde a su vez el hecho de usar hilos agiliza el tiempo de ordenamiento y envió de estos, un punto a destacar es que no siempre el uso de hilos y la división ayuda, ya que al tener demasiados hilos el crearlos genera un tiempo de demora interesante, por lo cual el uso de hilos genera un buen uso mientras que el abuso puede enfocar más los recursos hacia la creación de los hilos y generar más demora.

Luis Enrique Rojas Alvarado.

En esta practica se vio como el uso de los hilos a través de sockets para poder implementar ordenamientos en red, aunque también el hecho de usar demasiados hilos puede generar que la

maquina donde se creen los hilos se caiga por el uso de la RAM, además el uso excesivo de estos genera demoras.