Laboratório 2 - Programação para Sistemas Paralelos e Distribuídos

Paulo Vitor Coelho da Rocha - 17/0062465

TCP/IP significa protocolo de controle de transmissão/protocolo da internet (Transmission Control Protocol/Internet Protocol). TCP/IP é um conjunto de regras padronizadas que permitem que os computadores se comuniquem em uma rede como a internet.

O protocolo TCP é, talvez, o mais utilizado na camada de transporte para aplicações na Web. Diferente do UDP, o TCP é voltado à conexão e tem como garantia a integridade e ordem de todos os dados.

O protocolo UDP (sigla para *User Datagram Protocol*) tem, como característica essencial, um atributo que pode parecer esquisito para os iniciantes no tema - a **falta de confiabilidade.**

Isso significa que, através da utilização desse protocolo, pode-se enviar **datagramas** de uma máquina à outra, mas sem garantia de que os dados enviados chegarão intactos e na ordem correta.

Além do mais, o UDP é um protocolo que não é voltado à conexão. Isso significa que o "aperto de mão", ou, tecnicamente, ***handshake***, não é necessário para que se estabeleça uma comunicação.

Na primeira versão do código, foi desenvolvida uma solução onde não ocorria a paralelização do programa como podemos ver no código abaixo:

```c
int main(int argc, char *argv[])
{
        for (int i = 0; i < VECTOR_SIZE; i++)
        {
                vetorzinho.x[i] = pow(i - VECTOR_SIZE / 2, 2);
        }
        for (int i = 0; i < VECTOR_SIZE; i++)
        {
                vetorzinho.x[i] = sqrt(vetorzinho.x[i]);
        }

        struct sockaddr_in ladoServ; /* contem dados do servidor        */
        int sd;                                         /* socket descriptor              */
        int n, k;                                       /* num caracteres lidos do servidor */

        /* confere o numero de argumentos passados para o programa */
        if (argc < 3)
        {
                printf("uso correto: %s <ip_do_servidor> <porta_do_servidor>\n", argv[0]);
                exit(1);
        }

        memset((char *)&ladoServ, 0, sizeof(ladoServ)); /* limpa estrutura */

        ladoServ.sin_family = AF_INET;                          /* config. socket p. internet*/
        ladoServ.sin_addr.s_addr = inet_addr(argv[1]); /*ip*/
        ladoServ.sin_port = htons(atoi(argv[2]));          /*porta*/

        /* Cria socket */
        sd = socket(AF_INET, SOCK_STREAM, 0);
        if (sd < 0)
        {
                fprintf(stderr, "Criacao do socket falhou!\n");
                exit(1);
        }

        /* Conecta socket ao servidor definido */
        if (connect(sd, (struct sockaddr *)&ladoServ, sizeof(ladoServ)) < 0)
        {
                fprintf(stderr, "Tentativa de conexao falhou!\n");
                exit(1);
        }

        printf("> ");
        send(sd, &vetorzinho, sizeof(vetorzinho), 0); /* enviando dados ...  */
        recv(sd, &maxMin, sizeof(maxMin), 0);
        printf("Resposta do Sv: Max = %f and Min = %f\n", maxMin.max, maxMin.min);

        printf("------- encerrando conexao com o servidor -----\n");
        close(sd);
        return (0);
} /* fim do programa */
```

Diretamente na main, havia a criação do socket de comunicação, e também o envio e recebimento da mensagem de resposta do servidor com os valores máximos e mínimos.

Para a paralelização da solução, utilizei a biblioteca de threads da linguagem C, e com elas a main ficou um pouco mais complexa:

```c
int main(int argc, char *argv[])
{
        double time_spent = 0.0;

        clock_t begin = clock();

        struct sockaddr_in ladoServ; /* contem dados do servidor        */
        int sd;                                         /* socket descriptor             */
        int n, k;                                       /* num caracteres lidos do servidor */
        struct maxMinParcial maxMinParcial;

        for (int i = 0; i < VECTOR_SIZE; i++)
        {
                vetorzinho.x[i] = pow(i - VECTOR_SIZE / 2, 2);
        }
        for (int i = 0; i < VECTOR_SIZE; i++)
        {
                vetorzinho.x[i] = sqrt(vetorzinho.x[i]);
        }

        /* confere o numero de argumentos passados para o programa */
        if (argc < 5)
        {
                printf("uso correto: %s <ip_do_servidor> <porta_do_servidor> <ip_do_outro_servidor> <porta_do_outro_servidor>\n", argv[0]);
                exit(1);
        }

        struct ipPorta *ipPorta;
        ipPorta = malloc(sizeof(struct ipPorta));
        strcpy((*ipPorta).ip, argv[1]);
        strcpy((*ipPorta).porta, argv[2]);

        pthread_t thread1, thread2;
        pthread_create(&thread1, NULL, func, (void *)ipPorta);
```

A declaração do vetor inicial passou a ser feita de forma global e o seu preenchimento ficou dentro da main.

Após o preenchimento do vetor, criamos uma estrutura chamada ipPorta para passar o ip e a porta recebidos na hora da execução do programa, realizamos a criação da thread e ela fica responsável por se comunicar com um dos IP’s passados nos parametros iniciais. Para isso

temos a seguinte função:

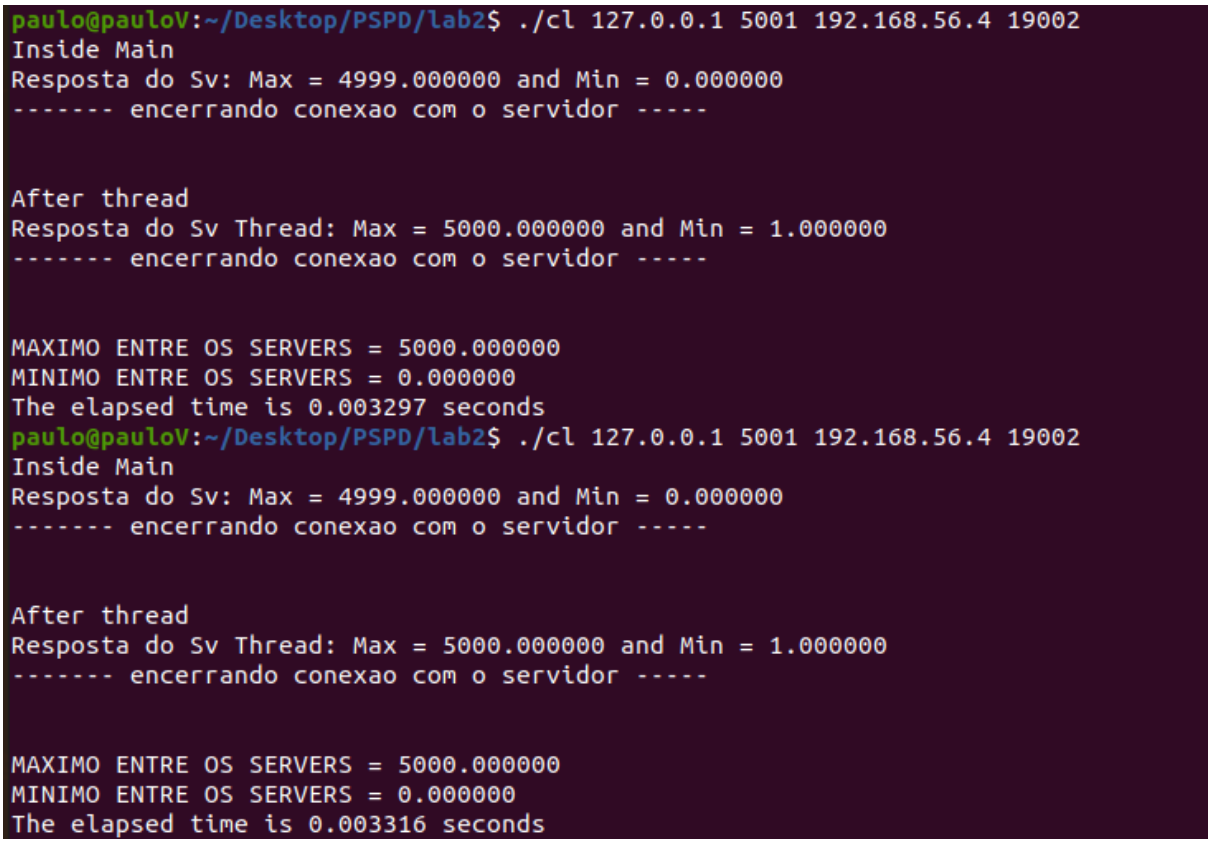
```c
void *func(void *arg)
{

        struct ipPorta *ipPorta = (struct ipPorta *)arg;

        struct sockaddr_in ladoServ1; /* contem dados do servidor       */
        int sd;                                         /* socket descriptor             */
        int n, k;                                       /* num caracteres lidos do servidor */

        memset((char *)&ladoServ1, 0, sizeof(ladoServ1)); /* limpa estrutura */

        ladoServ1.sin_family = AF_INET;                                         /* config. socket p. internet*/
        ladoServ1.sin_addr.s_addr = inet_addr(ipPorta->ip); /*ip*/
        ladoServ1.sin_port = htons(atoi(ipPorta->porta));       /*porta*/

        /* Cria socket */
        sd = socket(AF_INET, SOCK_STREAM, 0);
        if (sd < 0)
        {
                fprintf(stderr, "Criacao do socket falhou!\n");
                exit(1);
        }

        /* Conecta socket ao servidor definido */
        if (connect(sd, (struct sockaddr *)&ladoServ1, sizeof(ladoServ1)) < 0)
        {
                fprintf(stderr, "Tentativa de conexao falhou no athread!\n");
                exit(1);
        }
        struct maxMinParcial maxMinParcial;

        send(sd, &vetorzinho, sizeof(vetorzinho), 0); /* enviando dados ...  */
        recv(sd, &maxMinParcial, sizeof(maxMinParcial), 0);
        maxMin.max[1] = maxMinParcial.max;
        maxMin.min[1] = maxMinParcial.min;


        close(sd);


        pthread_exit(0);
}
```

Em cada uma delas, fazemos a instanciação do cliente de acordo com o IP’s dos hosts passado como argumento e a partir daí cada thread realiza sua função específica, nesse caso, calcular o maior e o menor valor de cada metade do vetor e coloca esse valor na estrutura de dados criada para receber o valor de cada um dos workers, como podemos ver na imagem abaixo:

```c
#define MAX_WORKERS 10
#define VECTOR_SIZE 10000

struct vetorDeFloats
{
        float x[VECTOR_SIZE];
} vetorDeFloats;

struct maxMinRes
{
        float max[MAX_WORKERS];
        float min[MAX_WORKERS];
} maxMinRes;

struct maxMinParcial
{
        float max;
        float min;
} maxMinParcial;

struct ipPorta
{
        char ip[100];
        char porta[20];
} ipPorta;

struct vetorDeFloats vetorzinho;
struct maxMinRes maxMin;
```

Podemos ver o programa funcionando, após ligar os servidores cada um em sua devida máquina através do comando:

tempo obtido sem paralelizar a solução.



tempo e output obtido paralelizando a solução.

Como podemos observar, a paralelização da solução trouxe um aumento no tempo de execução do programa tendo em vista a simplicidade e tamanho do vetor que precisava ser analisado.

Por indispor de tantas máquinas como havia pedido o enunciado, não foi realizado o experimento com mais de 2 máquinas, porém o código foi feito de uma maneira que para utilizar de máquinas adicionais, basta-se criar novas threads e passar o IP para cada uma delas. O protocolo de comunicação utilizado nos experimentos foi o TCP

por haver a necessidade de garantia de chegada da mensagem, fazendo-se necessário o estabelecimento de conexão (h*andshake*) entre as máquinas.

Opinião geral:

Trabalho muito interessante que desenvolve habilidades tanto de paralelismo com threads, como comunicação entre cliente/servidor com mais de um servidor (que geralmente não é tratada nas matérias anteriores).

Não tive dificuldades para realizar o experimento pois já tinha tido contato com conexão TCP na matéria de Fundamentos de Rede e Fundamentos de Sistemas Embarcados.

Fiz o experimento sozinho, portanto minha participação foi 100%.

Meu aprendizado acredito ter sido relativamente baixo por já conhecer o funcionamento dessa tecnologia, porém foi uma boa revisão sobre esse tipo de comunicação.

Nota sobre conhecimentos em comunicação TCP/IP e UDP: 9.5