Entrega da tarefa em MPI

A Maratona de Programação Paralela de 2019 disponibilizou diversos problemas para serem resolvidos em paralelo e/ou distribuído: http://lspd.mackenzie.br/marathon/19/problems.html. Escolha um daqueles problemas e, a partir do código-fonte e arquivos de entrada disponíveis, implemente uma solução em MPI.

Além de entregar o código-fonte da solução em paralelo, entregue também um documento descrevendo seu algoritmo paralelo, uma tabela contendo a medição de desempenho e os gráficos de escalabilidade forte.

Exercício em dupla.

Alessandro Bezerra Da Silva - 41908767
Pedro Unello Neto - 41929713

Programa escolhido - Password Cracking

O algoritmo desenvolvido se baseia na ideia, por mais que trivial, de dividir os caracteres passiveis de serem combinados entre os processos, para que, em um determinado momento e tamanho de combinação, cada processo tentara combinações que comecem com a letra que lhe fora dada, além disso o algoritmo foi engendrado de tal maneira que um overflow de processos, para o tamanho da cadeia de caracteres, fará com que os processos excedentes já comecem a trabalhar em um tamanho de combinação maior (logo para 36 caracteres, um 37 processo, tido que o rank 37 não é o que defini quem irá ser o excedente, iria diretamente começar a fazer combinações de 2 caracteres, começando com a letra a, e em um segundo dado momento iria fazer combinações de 2 caracteres também começando com a letra A), isto claro tentando evitar ao máximo que processos trabalhem simultaneamente nas mesmas combinações.

Logo o código consiste em:

1. Um processo lê os dados e manda por broadcast a hash procurada para todos
2. Para um limite lenMax / (1 + (Numero de processos / Numero de caracteres)), enquanto não achar a hash procurada faça
    a. Enquanto não iterar sobre o numero de caracteres ou achar a hash procurada faça
        i. Distribua uma letra para cada processo, com Scatter
        ii. Reatribua o tamanho da combinação que será iterada, de acordo com o índice do processo em relação a quantidade de caracteres combináveis (Se passar de tal quantidade, incrementa proporcionalmente)
        iii. Chame o procedimento recursivo para gerar todas as combinações, tendo a primeira letra garantida pela dis tribuição acima
        iv. Utiliza o Allreduce para achar o maior valor entre os *ok, assim garantindo que todos os processos saibam se alguém achou a hash (para aquela iteração)
        v. Incrementa o contador de letras iteradas com a quantidade de processos, e utiliza do operador modulo na quantidade de caracteres combináveis, para manter dentro de um limite de caracteres combinaveis

password_bf.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <openssl/md5.h>
#include <string.h>

#include <mpi.h>

#define MAX 10

#define MAIOR(a,b) a > b ? a : b

typedef unsigned char byte;

char letters[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890";

/*
 * Print a digest of MD5 hash.
*/
void print_digest(byte * hash){
    int x;

    for(x = 0; x < MD5_DIGEST_LENGTH; x++)
            printf("%02x", hash[x]);
    printf("\n");
}

/*
 * This procedure generate all combinations of possible letters
*/
void iterate(byte * hash1, byte * hash2, char *str, int idx, int len, int
*ok) {
    int c;

    // 'ok' determines when the algorithm matches.
    if(*ok) return;

    if (idx < (len - 1)) {
        // Iterate for all letter combination.
        for (c = 0; c < strlen(letters) && *ok==0; ++c) {
            str[idx] = letters[c];
            // Recursive call
            iterate(hash1, hash2, str, idx + 1, len, ok);
        }
    } else {
        // Include all last letters and compare the hashes.
```

```
        for (c = 0; c < strlen(letters) && *ok==0; ++c) {
            str[idx] = letters[c];
            MD5((byte *) str, strlen(str), hash2);
            if(strncmp((char*)hash1, (char*)hash2, MD5_DIGEST_LENGTH) ==
0){
                printf("found: %s\n", str);
                //print_digest(hash2);
                *ok = 1;
            }
        }
    }
}

/*
 * Convert hexadecimal string to hash byte.
*/
void strHex_to_byte(char * str, byte * hash){
    char * pos = str;
    int i;

    for (i = 0; i < MD5_DIGEST_LENGTH/sizeof *hash; i++) {
        sscanf(pos, "%2hhx", &hash[i]);
        pos += 2;
    }
}

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

    MPI_Init(NULL, NULL);

    // Numero de processos
    int nProcessos;
    MPI_Comm_size(MPI_COMM_WORLD, &nProcessos);
    // Rank do processo
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    char str[MAX+1];
    int len;
    int lenMax = MAX;
    int ok = 0, r;
    char hash1_str[2*MD5_DIGEST_LENGTH+1];
    byte hash1[MD5_DIGEST_LENGTH]; // password hash
    byte hash2[MD5_DIGEST_LENGTH]; // string hashes

    int* indices;

    if (rank == 0 ){ //Apenas o rank zero faz a leitura
```

```
        // Input:
        r = scanf("%s", hash1_str);

        // Check input.
        if (r == EOF || r == 0)
        {
            fprintf(stderr, "Error!\n");
            exit(1);
        }

        // Convert hexadecimal string to hash byte.
        strHex_to_byte(hash1_str, hash1);

        //Aloca um vetor de indices na quantidade de processos, e atribui
o proprio indice
        indices = malloc(sizeof(int) * nProcessos);
        for (int i = 0; i < nProcessos; i++){ *(indices + i) = i; }

    }

    //Feita a leitura, o rank zero espalha por broadcast a hash a ser
encontrada
    MPI_Bcast(hash1, MD5_DIGEST_LENGTH, MPI_BYTE, 0, MPI_COMM_WORLD);

    memset(hash2, 0, MD5_DIGEST_LENGTH);
    //print_digest(hash1);


    int count = 0; //Contador de caracteres passiveis de combinacao
(vetor letters)
    /*Tendo que cada passo deveria fazer combinacoes com 36 caracteres,
incrementando o tamanho das combinacoes
    *divide/seciona a quantidade de passos dependendo do numero de
processos, afinal se existem, ao exemplo, 72 processos
    para 36 letras, nao seria preciso de duas iteracoes para dividir 72
letras em 1 letra por processo*/
    for(len = 1; len <= lenMax && ok == 0; len += 1 +
((int)(nProcessos/36))){

        //A operacao de modulo garante (ou pelo menos tenta garantir),
que nao serao repetidas letras
        count %= 36;

        ///enquanto nao for achado a resposta e nao forem feitas n de len
combinaçoes para cada um dos caracteres
        while (count < 36 && ok == 0) {

            memset(str, 0, len+1);
```

```
            //divide as letras em n processos

            int cur;

            //utilizando do scatter, para distribuir os indices antes
alocados pelo rank 0
            MPI_Scatter(indices, 1, MPI_INT, &cur, 1, MPI_INT, 0,
MPI_COMM_WORLD);

            //Cada processo pega uma letra inicial para as combinacoes
            *str = letters[(count + cur)%36];

            /*tendo que a qtd de processos pode exceder a quantidade de
caracteres
            *para os processos com rank excedente dessa qtd, o tamanho da
combinacao atual
            sera equivalentemente incrementado*/
            int curLen = len + ((int)(cur/36));
            //Se ainda estiver no limite de lenMax, chama a iteracao com
o primeiro indice preenchido
            if ( curLen <= lenMax) { iterate(hash1, hash2, str, 1,
curLen, &ok); }

            //Utiliza a funcao de maximo no reduce para descobrir se
alguem achou a senha,
            //neste caso o Allreduce e utilizado pois tal condicao devera
ser espalha entre os processos
            MPI_Allreduce(&ok, &ok, 1, MPI_INT, MPI_MAX, MPI_COMM_WORLD);

            count+= nProcessos;
        }
    }

    if ( rank == 0) { free(indices); }

    MPI_Finalize();

    return 0;

}
```

B (shell script ?)

O arquivo B disponibilizado foi aproveitado para gerar os resultados de speedup e escalabilidade, já que nele foi realizado um laço para uma quantidade arbitraria de

processos, tais que levam a medição do tempo de todas as execuções, da serial até MPI com {quantidade de processos} passada, todos os resultados são inseridos nos arquivos de texto, como visto abaixo.

```
#!/bin/bash
#Este arquivo roda o programa N vezes, com um numero incremental de processos em MPI, e coloca tais resultados nos arquivos, como dito abaixo
(time ./serial < password.in) > SerialResultado.txt 2> TempoSerial.txt
export MPI_MAX_PROCESS=$1
for MPI_PROCESS in $(seq 2 $MPI_MAX_PROCESS)
do
    echo "Processos = " $MPI_PROCESS >> TempoMPI.txt
    (time mpirun -host localhost:$MPI_PROCESS -np $MPI_PROCESS ./password_bf < password.in) >> MPIResultado.txt 2>> TempoMPI.txt
done
```

makefile

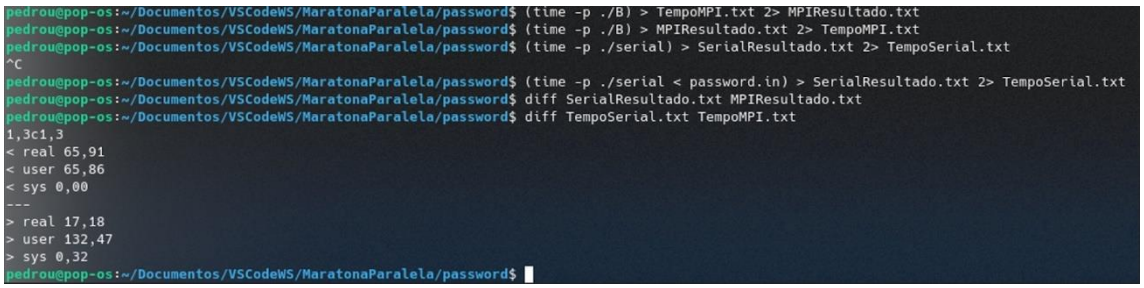
A única diferença aqui realizada foi a inclusão da metodologia de compilação em ambos os casos (serial e MPI).

```
FLAGS=-O3 -lssl -lcrypto
CC=gcc
MCC=mpicc
RM=rm -f

EXECs=serial
EXECp=password_bf

all: $(EXECp) $(EXECs)

$(EXECp):
    $(MCC) $(FLAGS) $(EXECp).c -c -o $(EXECp).o
    $(MCC) $(EXECp).o -o $(EXECp) $(FLAGS)

$(EXECs):
    $(CC) $(FLAGS) $(EXECs).c -c -o $(EXECs).o
    $(CC) $(EXECs).o -o $(EXECs) $(FLAGS)

run:
    time mpirun -host localhost:8 -np 8 ./$(EXEC) < password.in

clean:
    $(RM) $(EXECs).o $(EXECs)
    $(RM) $(EXECp).o $(EXECp)
```

password.in

```
afa345bc5ced1b9bf90a3ff76d8ac111
```

Execução

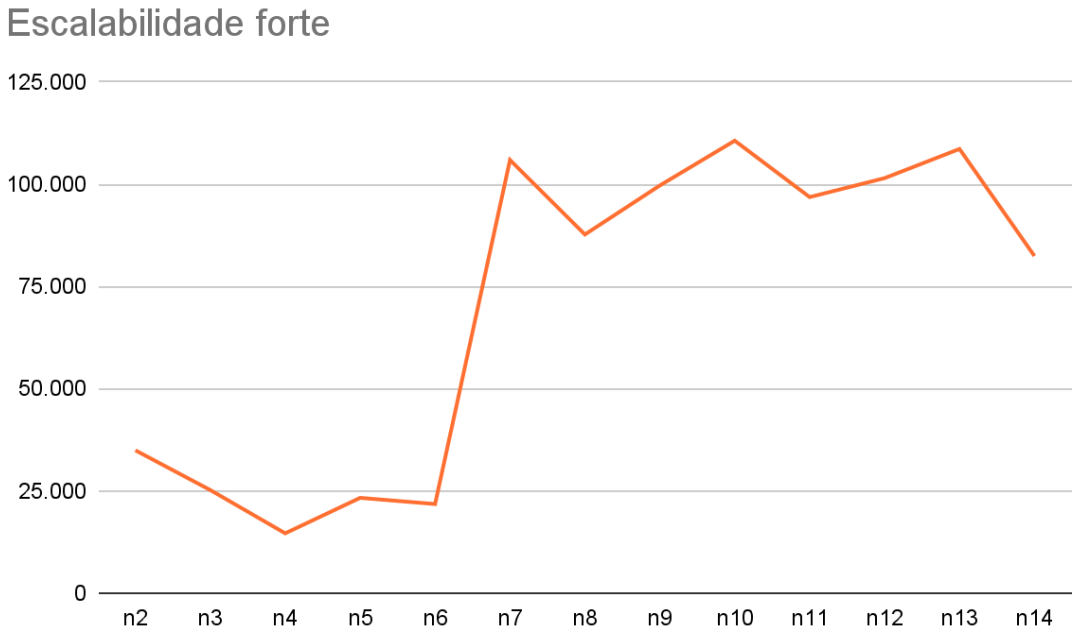


*Serial 65 .91
Paralelo 17.18 - 4 processo

SpeedUp = 65.91/17.18 => 3,836437718277066

Tabela
*Serial 65 .91

Processos	Tempo(s)	Speedup
n2	35	1.887294906
n3	25	2.614231318
n4	15	4.504202829
n5	23	2.827663134
n6	22	3.024643201
n7	106	0.6221974682
n8	88	0.7514964939
n9	100	0.6611495636
n10	111	0.5956835313
n11	97	0.6805298861
n12	101	0.6494363866
n13	109	0.6067999153
n14	82	0.7993257092


Escalabilidade forte
125.000
100.000
75.000
50.000
25.000
0
n2
n3
n4
n5
n6
n7
n8
n9
n10
n11
n12
n13
n14