PEDRO IVO FRAGA

RODRIGO SILVEIRA

PATTERN MATCHING EM MPI

Trabalho apresentado no Curso de Ciências da Computação, Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS).

Professor: Tiago Ferreto

PORTO ALEGRE, RIO GRANDE DO SUL

2018

1. INTRODUÇÃO

Este relatório visa demonstrar como foi o processo de desenvolvimento de um algoritmo de *Wildcard Pattern Matching* com programação paralela em MPI, no escopo da disciplina de Programação Paralela. Para o desenvolvimento do mesmo, utilizamos a linguagem de programação C++. O objetivo final deste estudo é demonstrar a diferença de desempenho entre um algoritmo comum e um algoritmo paralelizado.

1.2. O que é o *Wildcard Pattern Matching*?

O *Wildcard Pattern Matching* é considerado um problema de casamento de padrões, onde visa-se encontrar se uma certa cadeia de caracteres pertence a outra cadeia de caracteres, normalmente um texto ou uma frase, ou algo similar.
Comumente, o algoritmo recebe como entrada a cadeia de caracteres que deve ser encontrada e a cadeia de caracteres considerada como o "texto" no qual deve-se procurar a primeira cadeia, assim, retornando um valor booleano de verdadeiro ou falso, caso seja encontrada a cadeia de caracteres ou não. Porém, no escopo deste trabalho, trabalharemos com um algoritmo que retorna o número de vezes que a primeira cadeia de caracteres é encontrada dentro da segunda.

1.3 O que é MPI?

MPI vem do inglês, *Message Passing Interface* (Interface para Transferência de Mensagem, em português), e é uma API (interface de programação de aplicativo) para C, C++ e várias outras linguagens que permite acrescentar simultaneidade aos programas, tornando-os paralelos.

2. O ALGORITMO

O algoritmo desenvolvido consiste em receber como entrada uma cadeia de caracteres denominada de "*pat"* (*pattern*) e uma cadeia de caracteres denominada de "*txt*" (*text*). O algoritmo visa encontrar e retornar quantas vezes a cadeia *pat* pode ser encontrada dentro da cadeia *txt*. Para isso, algumas regras de equivalência foram definidas, e o algoritmo foi desenvolvido de forma que possui um algoritmo principal com um algoritmo auxiliar, ambos com alguns laços de repetição.

2.1. Caso Padrão

O caso padrão consiste em comparar a cadeia com total equivalência, possuindo como entrada o *pat* “ab”, por exemplo, e o *txt* “aaaababbbb”. O algoritmo deve retornar 2, pois o *pat* pode ser encontrado duas vezes dentro do *txt* “aaa**abab**bbb”.

2.2. Primeiro Coringa

O primeiro coringa a ser considerado é o caractere “**?**”, que equivale a qualquer outro caractere. Por exemplo, a cadeia de entrada *pat* “a?b” ocorre uma vez dentro da cadeia de entrada *txt* “**aab**bba”.

2.3. Segundo Coringa

Já o segundo coringa que deve ser considerado é o caractere “*****”, que equivale a qualquer caractere (incluindo vazio) inúmeras vezes. Com a cadeia de entrada *pat* “a*b”, o algoritmo retorna o valor 1 para a cadeia de entrada *txt* “aaaaaaaaaab”.

2.4. O código

O código consiste em dois métodos, um o método padrão e o outro um método de auxílio, que realiza a chamada para esse método padrão (tanto o paralelizado quanto o sequencial).

```
const int intervalo = 5000;
double starttime, stoptime;

char filepath[150];
strcpy(filepath, argv[2]);
cout << "Read file " << filepath << endl;

int row = 0;
int count = 0;
std::ifstream file(filepath);
std::string line;
std::stringstream l;
while (std::getline(file, line)) {
    l << line;
}


starttime = omp_get_wtime();

count = countFreq(pattern, l.str().c_str());

cout << "count: " << count << endl;
stoptime = omp_get_wtime();
printf("Tempo de execução: %3.2f segundos\n", stoptime-starttime);
```

```
int countFreq(const char* pat, const char* txt)
{
    int res = 0;
    int lenPat = strlen(pat);
    int lenTxt = strlen(txt);
    int idxPat, idxIni, idxTxt;
    bool bOk = false;
    bool bCoringa = false;
    for (idxIni = 0; idxIni < lenTxt; idxIni++) {

        idxPat = 0;
        idxTxt = idxIni;
        bOk = false;

        for (idxPat = 0; idxPat < lenPat; ) {
            if (idxTxt >= lenTxt) break;
            if (pat[idxPat] == txt[idxTxt] || pat[idxPat] == '?'){
                bOk = true;
                bCoringa = false;
                idxPat++;
                idxTxt++;
            } else {
                if (bCoringa) {
                    idxTxt++;
                }
                else if (pat[idxPat] == '*'){
                    bCoringa = true;
                    idxPat++;
                } else {
                    bOk = false;
                    break;
                }
            }
        }
        if (bOk && idxPat == lenPat) {
            res++;
        }

    }

    return res;
}
```

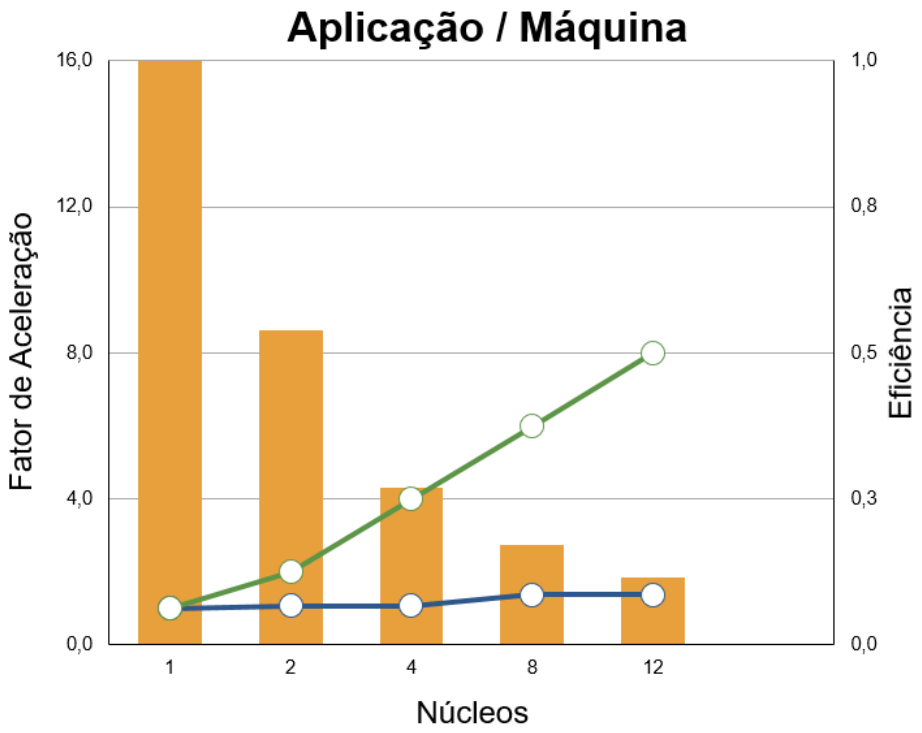
3. A PARALELIZAÇÃO

Com o auxílio do MPI, a paralelização se deu forma mais complicada do que com OpenMP (API utilizada no primeiro trabalho da disciplina), adicionando várias linhas de código, com auxílio de vetores que são processados de forma diferente em cada um dos processos que o MPI gera.

```
MPI_Status status;
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
double t1,t2;
t1 = MPI_Wtime();  // inicia a contagem do tempo
int chunksize = VETSIZE/size;
if(rank == 0){
    while (std::getline(file, line)) {
        vetor[rows] << line;
        rows++;
    }
    //envia chunk pra cada processo
    for(int i=1; i<size; i++) {
        int begin = i*chunksize;
        MPI_Send(&vetor[begin], chunksize, MPI_INT, i, 100, MPI_COMM_WORLD);
    }
    // Mestre processa chunk local
    for(int i=0; i<chunksize; i++) {
        resultado[i] = countFreq(pattern, vetor[i].str().c_str());
    }
    // Recebe respostas dos outros processos
    for(int i=1; i<size; i++) {
        MPI_Recv(auxbuf, chunksize, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
        int source = status.MPI_SOURCE;
        int begin = source*chunksize;
        memcpy(&resultado[begin], auxbuf, chunksize*sizeof(int));
    }
}
else{
    // Recebe chunk
    MPI_Recv(vetor, chunksize, MPI_INT, 0,
    MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    // Processa chunk
    for(int j=chunksize*rank; j<(chunksize*(rank+1))-1; j++) {
        resultado[j] = countFreq(pattern, vetor[j].str().c_str());
    }
    // Envia respostas
    MPI_Send(resultado, chunksize, MPI_INT, 0,
    100, MPI_COMM_WORLD);
}

if(rank == 0) {
    for(int i=0; i<VETSIZE; i++) {
        count += resultado[i];
    }
    t2 = MPI_Wtime(); // termina a contagem do tempo

    cout << "count: " << count << endl;
    printf("\nTempo de execucao: %f\n\n", t2-t1);

    MPI_Finalize();
}
```

4. O DESEMPENHO

Infelizmente, devido às dificuldades de acesso ao *cluster*, ocorreram poucos testes e o desenvolvimento do trabalho se deu de forma curta, apenas nos momentos em que os alunos poderiam acessar o marfim diretamente da PUCRS e não estavam em aula. Logo, o tempo de desenvolvimento foi relativamente curto para o que gostaríamos de apresentar. Porém, obtivemos alguns resultados que são representados na imagem abaixo.



5. CONSIDERAÇÕES FINAIS

Novamente, como no primeiro trabalho, voltamos a enfrentar a dificuldade de desenvolvido devido ao fato de se necessitar acessar o Cluster Marfim, que se dá de forma horrível de fora da PUCRS. Porém, podemos concluir que a programação com o auxílio da API do MPI é realmente complexa, necessitando de bastante prática para se aperfeiçoar. Apesar de seus defeitos, ela é muito útil, tornando programas com altos gargalos em programas com desempenho relativamente melhores. Infelizmente, nossos testes não conseguem demonstrar isto com tanta precisão, porém podemos observar uma melhora no desempenho em relação ao OpenMP.