Pontifícia Universidade Católica de Minas Gerais

Campus Poços de Caldas

Curso de Ciência da Computação

Algoritmo para a busca de ocorrências em uma sequência utilizando computação paralela para otimizar o tempo de busca.

Jean Antonio Martins[1]

Ulysses Henrique Ferreira[2]

04/2023

[1] Estudante de Graduação do 8º período do Curso de Ciência da Computação da PUC Minas; e-mail: jean.antonio@sga.pucminas.br

[2] Estudante de Graduação do 8º período do Curso de Ciência da Computação da PUC Minas; e-mail: uferreira@sga.pucminas.br

OBJETIVO:

O presente trabalho tem como objetivo fazer a comparação da velocidade de execução entre o algoritmo sequencial que busca a quantidade de ocorrências de uma determinada sequência de caracteres em um arquivo com sua versão paralela e plotar os resultados em um gráfico para melhor análise.

Também será feito o cálculo de speedup e eficiência para que seja possível analisar o desempenho da execução paralela e concluir se houve ou não ganho de performance e a partir de que ponto essa melhora começa, caso houver.

DESCRIÇÃO DO ALGORITMO SEQUENCIAL:

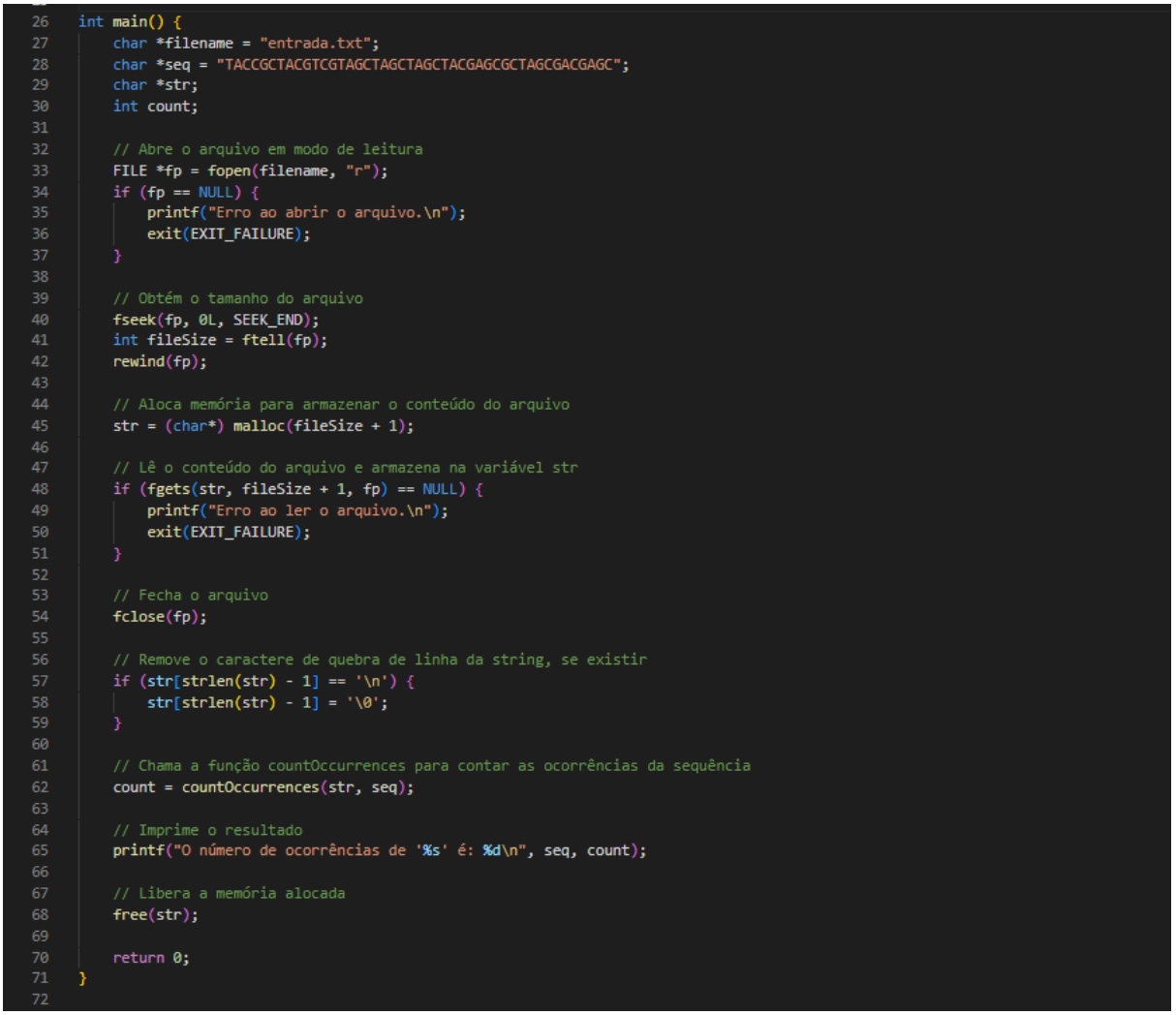
O código conta com um laço de repetição que lê de um arquivo os dados desde o início até o final, já desconsiderando a possível presença da sequência no final do arquivo. Dentro do loop principal, é realizada uma segunda iteração para verificar se uma sequência ocorre naquela posição do texto. Quando encontrada a sequência, o contador é incrementado e o loop interno é reiniciado.

DESCRIÇÃO DO CÓDIGO SEQUENCIAL:

O código é iniciado com a inicialização das variáveis junto com a abertura do arquivo de texto onde será feita a busca do número de ocorrências de determinada sequência, fazendo também a verificação de possíveis erros na abertura do arquivo, e encerrando a aplicação caso haja algum. Em seguida é utilizada a função "fseek", que reposiciona o indicador de posição do arquivo para uma determinada posição, neste caso para o final, possibilitando obter o tamanho total do arquivo a partir da função "ftell", que retorna a posição atual do ponteiro. Após obter o tamanho total do arquivo, é utilizada a função rewind para posicionar o ponteiro do arquivo de volta para o início.

Então, a memória é alocada de acordo com o tamanho do arquivo obtido previamente, para que seja possível armazenar o conteúdo do texto em uma variável, e para isso usamos a função fgets.

Figura 1 - Função main() na versão sequencial

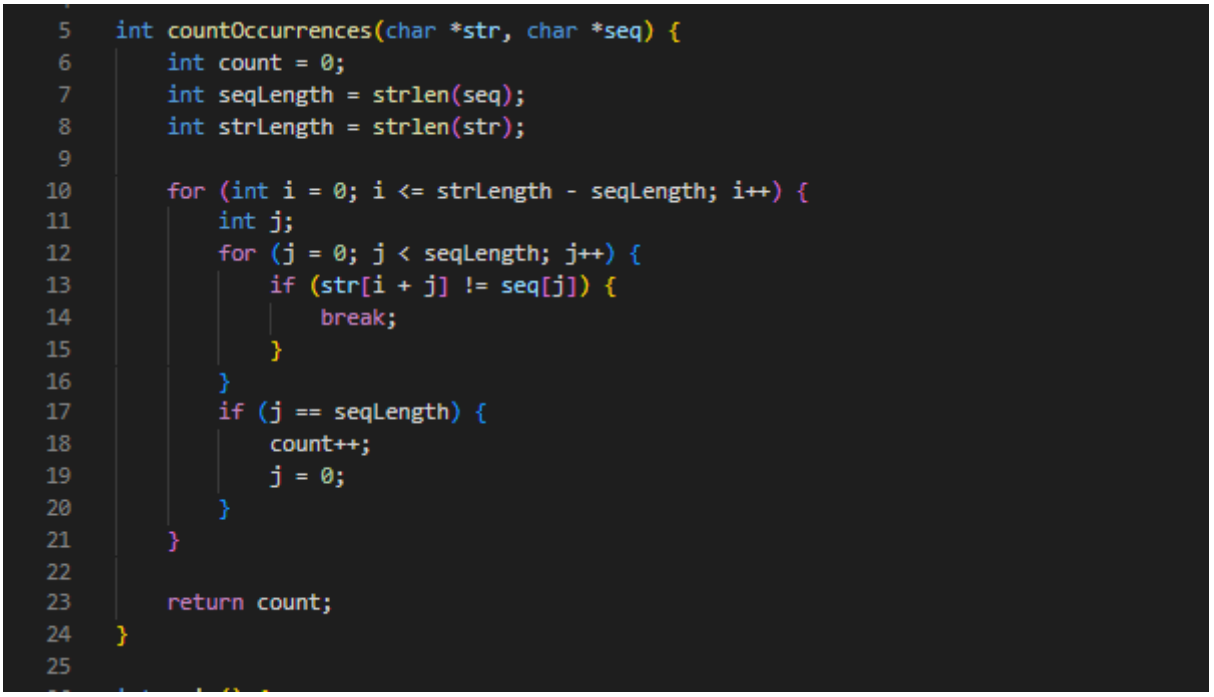


```c
int main() {
    char *filename = "entrada.txt";
    char *seq = "TACCGCTACGTCGTAGCTAGCTAGCTACGAGCGCTAGCGACGAGC";
    char *str;
    int count;

    // Abre o arquivo em modo de leitura
    FILE *fp = fopen(filename, "r");
    if (fp == NULL) {
        printf("Erro ao abrir o arquivo.\n");
        exit(EXIT_FAILURE);
    }

    // Obtém o tamanho do arquivo
    fseek(fp, 0L, SEEK_END);
    int fileSize = ftell(fp);
    rewind(fp);

    // Aloca memória para armazenar o conteúdo do arquivo
    str = (char*) malloc(fileSize + 1);

    // Lê o conteúdo do arquivo e armazena na variável str
    if (fgets(str, fileSize + 1, fp) == NULL) {
        printf("Erro ao ler o arquivo.\n");
        exit(EXIT_FAILURE);
    }

    // Fecha o arquivo
    fclose(fp);

    // Remove o caractere de quebra de linha da string, se existir
    if (str[strlen(str) - 1] == '\n') {
        str[strlen(str) - 1] = '\0';
    }

    // Chama a função countOccurrences para contar as ocorrências da sequência
    count = countOccurrences(str, seq);

    // Imprime o resultado
    printf("O número de ocorrências de '%s' é: %d\n", seq, count);

    // Libera a memória alocada
    free(str);

    return 0;
}
```

Fonte: De autoria própria

Com uma variável armazenando o texto é possível chamar a função "countOcurrences", que percorre todo o texto em um loop principal e em um loop interno que executa de acordo com o tamanho da sequência procurada. Se o loop interno comparar dois caracteres que não coincidem com a sequência, ele é interrompido e logo após há uma verificação que avalia se a variável de controle do loop interno foi incrementada ao máximo (tamanho total da sequência), se foi então há uma ocorrência e a variável de contagem é incrementada e, caso contrário, a contagem não é alterada e o loop externo dá continuidade repetindo o processo descrito até o final do arquivo. E ao fim da execução da função, é retornado o número total de ocorrências encontradas para a função main.

Figura 2 - Função countOcurrences() na versão sequencial



```c
int countOccurrences(char *str, char *seq) {
    int count = 0;
    int seqLength = strlen(seq);
    int strLength = strlen(str);

    for (int i = 0; i <= strLength - seqLength; i++) {
        int j;
        for (j = 0; j < seqLength; j++) {
            if (str[i + j] != seq[j]) {
                break;
            }
        }
        if (j == seqLength) {
            count++;
            j = 0;
        }
    }

    return count;
}
```

Fonte: De autoria própria

DESCRIÇÃO DO ALGORITMO PARALELO:

O código da busca de sequência será executado em todos os elementos do cluster, ficando cada nó responsável pela busca em uma subdivisão do texto original. A divisão é feita de forma que cada nó lê determinada parte do texto, sem repetir o mesmo trecho de outros EPs (elementos de processamento). O bloco de texto que cada nó recebe é definido a partir da divisão entre o número de caracteres do texto com a quantidade de EPs disponíveis, e caso haja resto nessa divisão o nó mestre se encarrega desses caracteres excedentes.

DESCRIÇÃO DO CÓDIGO PARALELO:

O código inicia com a declaração das variáveis necessárias e do MPI, que será responsável pela comunicação entre elementos de um cluster. A inicialização do ambiente de comunicação do MPI é realizada a partir da chamada da função MPI_Init.

As variáveis rank e size são definidas para armazenar o número referente ao nó atual e a quantidade de nós totais no cluster, respectivamente utilizando a função MPI_Comm_rank que retorna o rank do processo ativo. A variável rank será utilizada futuramente para definir o trecho de código que cada nó irá executar e também será usada no cálculo de divisão de blocos de caracteres.

O nome do arquivo é armazenado a partir do argumento passado na execução da aplicação no terminal, após o nome do executável. Por último, é definido a variável tempo, responsável por armazenar o tempo de execução do código em cada elemento do cluster, a partir da função MPI_Wtime(), que retornará o tempo em segundos desde um momento arbitrário no passado.

Em seguida o arquivo contendo o texto é lido e é obtido seu tamanho. Com seu tamanho e o valor atual de rank, é calculado o tamanho que o bloco que será analisado por cada nó terá, e caso não seja possível dividir igualmente entre os nós, o mestre fica encarregado da maior parte, sendo o tamanho do bloco somado ao resto da divisão.

O tamanho do bloco é utilizado para que seja feita a chamada da função "fseek" que posicionará o ponteiro do arquivo para a posição onde se inicia o bloco do nó que está executando e então é armazenado em uma variável o conteúdo.

Figura 3 - Função main() na versão paralela

```
30  int main(int argc, char **argv) {
31      int rank, size;
32      char *filename = argv[1];
33      char *seq = "TACCGCTACGTCGTAGCTAGCTAGCTACGAGCGCTAGCGACGAGC";
34      char *str;
35      int count;
36      double tempo = 0;
37      tempo = MPI_Wtime();
38
39      MPI_Init(&argc, &argv);
40      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
41      MPI_Comm_size(MPI_COMM_WORLD, &size);
42
43      // Abre o arquivo em modo de leitura
44      FILE *fp = fopen(filename, "r");
45      if (fp == NULL) {
46          printf("Erro ao abrir o arquivo.\n");
47          MPI_Finalize();
48          exit(EXIT_FAILURE);
49      }
50
51      // Obtém o tamanho do arquivo
52      fseek(fp, 0L, SEEK_END);
53      int fileSize = ftell(fp);
54      rewind(fp);
55
56      // Aloca memória para armazenar a parte do arquivo correspondente a este processo
57      int blockSize = fileSize / size; // pega qual vai ser o tamanho do bloco
58      if (rank == size - 1) {
59          blockSize += fileSize % size; // se for o mestre é o tamanho do bloco somado ao resto caso a divisao nao for exata
60      }
61      str = (char*) malloc(blockSize + 1); // aloca memoria de acordo com o tamanho bloco que o nó recebe
62
63      // Lê a parte do arquivo correspondente a este processo
64      fseek(fp, rank * blockSize, SEEK_SET);
65      if (fgets(str, blockSize + 1, fp) == NULL) {
66          printf("Erro ao ler o arquivo.\n");
67          MPI_Finalize();
68          exit(EXIT_FAILURE);
69      }
```

Fonte: De autoria própria

Com o bloco referente aquele nó armazenado, já é possível chamar a função "countOccurrences", que tem o mesmo comportamento da sua versão sequencial, diferenciando apenas na chamada da função MPI_Reduce, que combina valores em buffer de

cada EP e retorna o valor combinado para o nó mestre, obtendo dessa forma a contagem total de ocorrências encontradas no arquivo.

Figura 4 - Função countOcurrences() na versão paralela

```
int countOccurrences(char *str, char *seq, int size) {
    int count = 0;
    int seqLength = strlen(seq);
    int strLength = strlen(str);
    int i = 0;

    for (i; i <= strLength - seqLength; i++) {
        int j;
        for (j = 0; j < seqLength; j++) {
            if (str[i + j] != seq[j]) {
                break;
            }
        }
        if (j == seqLength) {
            count++;
            j = 0;
        }
    }

    int totalCount = 0;
    MPI_Reduce(&count, &totalCount, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    return totalCount;
}
```

Fonte: De autoria própria

ANÁLISE DOS RESULTADOS:

Para realizar a comparação de resultados entre o algoritmo sequencial e paralelo foi simulado a busca em um arquivo que contém uma sequência genética aleatória. Foi utilizado arquivos de teste contendo de 150.000 a 1.600.000.000 caracteres.

Nos gráficos gerados, o eixo x contém o número total de caracteres do arquivo e o eixo y o tempo em segundos gasto para realizar a busca.

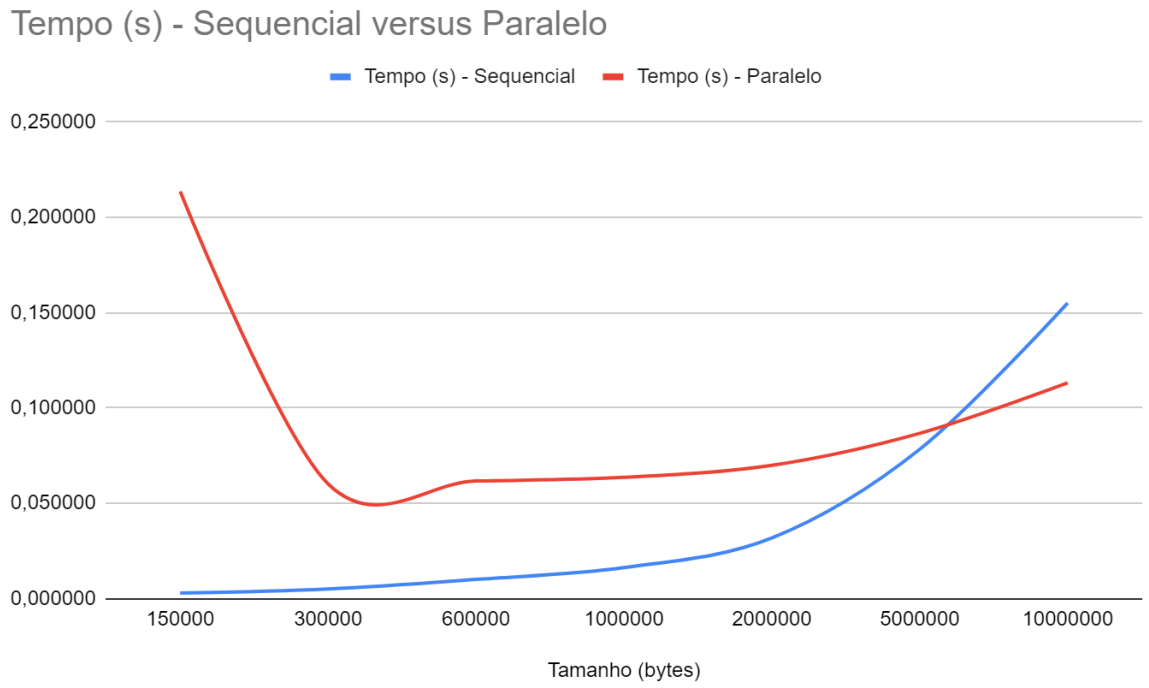


Figura 5 - Gráfico de eficiência para os primeiros valores.

No gráfico acima, é possível observar que para quantidades menores de caracteres a busca paralela não obtêm melhores resultados se comparado ao sequencial, porém pode-se notar que com o crescimento do arquivo, de que a execução do código paralelo obtém melhor performance.

A partir disso a diferença entre as duas execuções só aumenta com a versão paralela obtendo o melhor resultado, como é possível observar no seguinte gráfico:

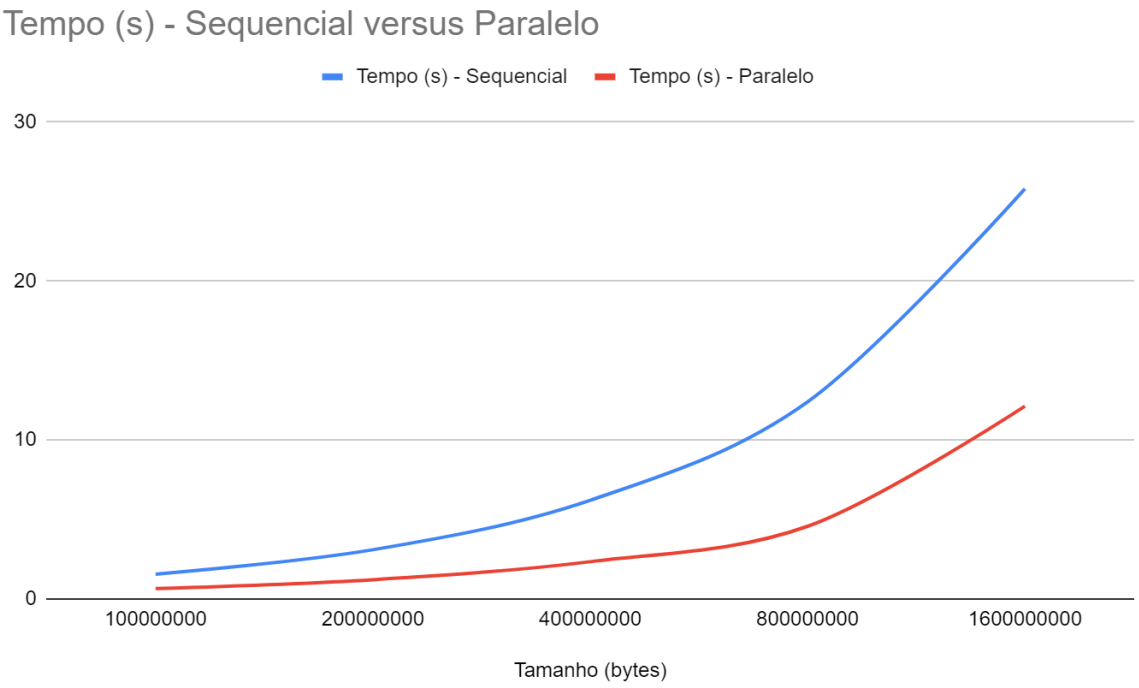


Figura 6 - Gráfico de eficiência para os valores restantes.

Tamanho (bytes)	Tempo (s) - Sequencial
150000	0,002785
300000	0,004963
600000	0,010000
1000000	0,016173
2000000	0,031748
5000000	0,078319
10000000	0,155148
100000000	1,532296
200000000	3,068590
400000000	6,188852
800000000	12,391370
1600000000	25,812307

Figura 7 - Tabela contendo o tempo de execução a partir do código sequencial.

Tamanho (bytes)	Tempo (s) - Paralelo	Speedup	Eficiência
150000	0,213733	0,0130	0,0043
300000	0,060131	0,0825	0,0275
600000	0,061655	0,1622	0,0541
1000000	0,063577	0,2544	0,0848
2000000	0,069914	0,4541	0,1514
5000000	0,086724	0,9031	0,3010
10000000	0,113179	1,3708	0,4569
100000000	0,623445	2,4578	0,8193
200000000	1,177747	2,6055	0,8685
400000000	2,314814	2,6736	0,8912
800000000	4,545400	2,7261	0,9087
1600000000	12,117388	2,1302	0,7101

Figura 8 - Tabela contendo o tempo de execução a partir do código paralelo.

Foi feito também outros dois gráficos, um para analisar o speedup obtido com o código paralelo e outro para sua eficiência.

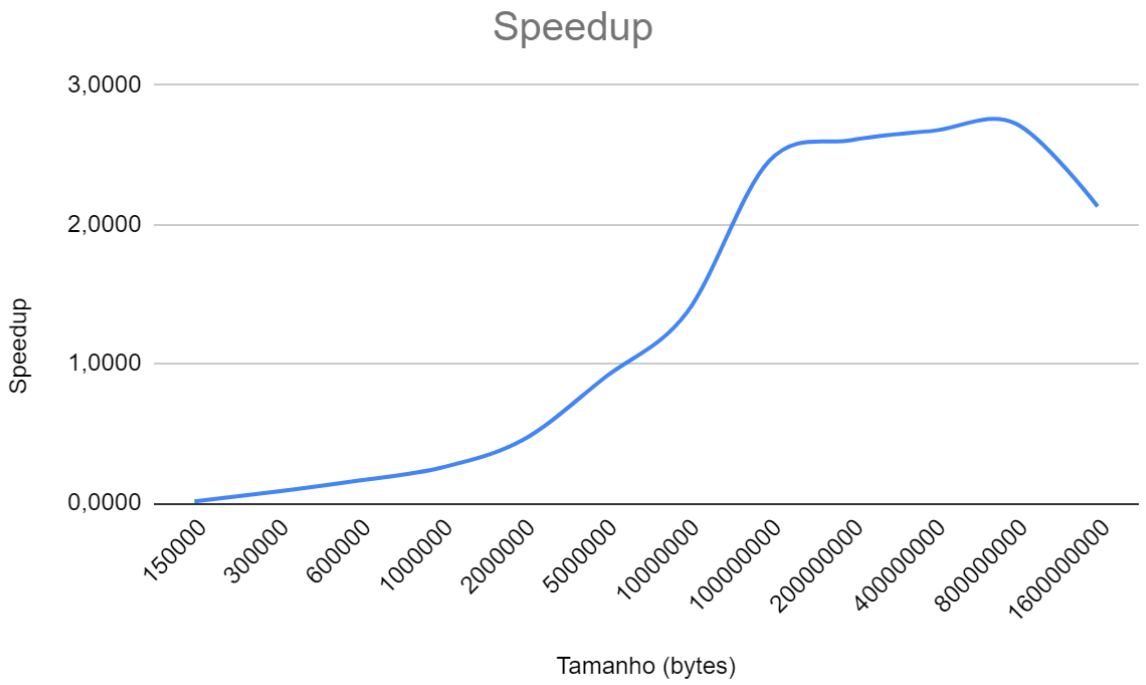


Figura 9 - Gráfico contendo o speedup.

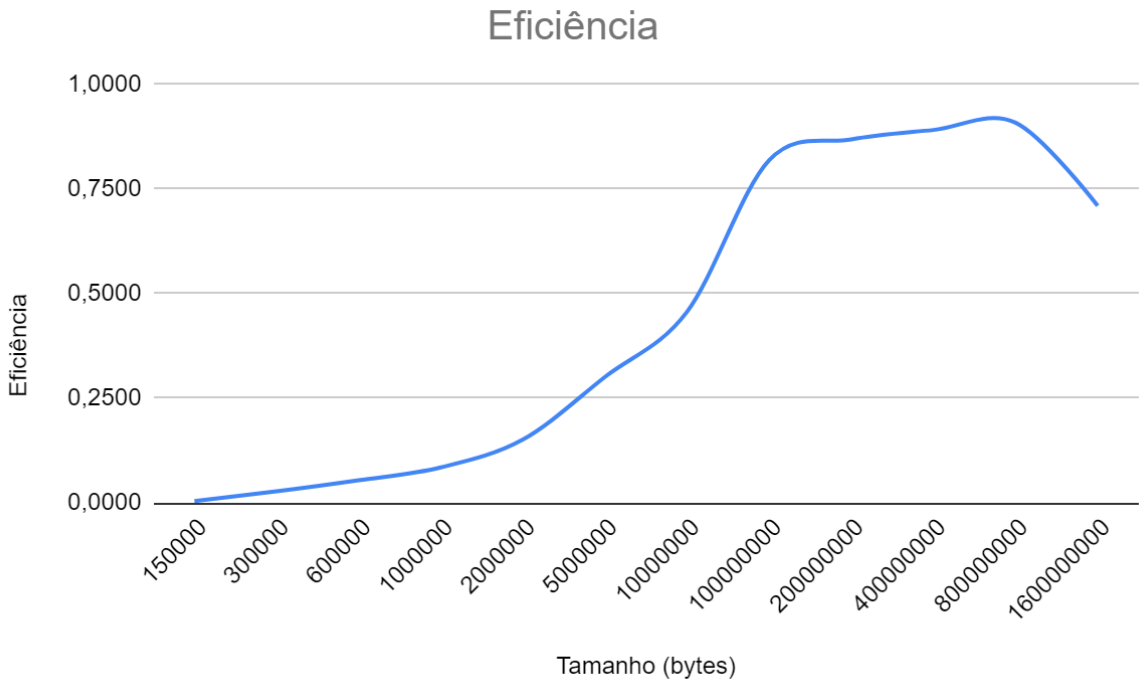


Figura 10 - Gráfico contendo a eficiência.

CONCLUSÕES:

Com base nos resultados obtidos, é possível concluir que inicialmente, para buscas em arquivos menores, não há ganho de performance na utilização de programação paralela, porém a partir de aproximadamente 5.000.000 caracteres, o desempenho do código paralelo se sobressai devido ao fato de que cada nó do cluster lê uma parte específica do arquivo, tornando o processamento dos dados menor, enquanto na versão sequencial, a leitura é realizada de ponta a ponta.

Também é de se notar o ganho de speedup e eficiência que aumenta proporcionalmente devido ao tamanho da entrada de dados, obtendo seu melhor desempenho na execução em que a entrada foi de 800.000.000 caracteres, onde houve speedup de 2,7261 segundos e uma eficiência de 90%. A partir de uma entrada de 1.600.000.000 de caracteres, com as configurações do cluster em que a análise foi realizada, é possível constatar uma queda de performance deduzindo ser devido ao elevado tamanho em bytes que cada nó está processando.

REFERÊNCIAS BIBLIOGRÁFICAS:

BACKES, A. Linguagem C - Completa e Descomplicada. 2 ed: GEN LTC, 2018.

JOSÉ, V. MPI: Uma ferramenta para implementação paralela, 2002. Disponível em: <https://www.scielo.br/j/pope/a/mLv97ppcXCbVYbv9JHgkK4B>. Acesso em 26 abr. 2023.