UNIVERSIDADE PRESBITERIANA MACKENZIE

Computação Paralela [Turma 05G11]

Enzo Guarnieri - 10410074

Erika Borges Piaui - 10403716

Júlia Campolim de Oste - 10408802

Otávio Augusto Freire de Andrade Bruzadin - 10409053

Projeto – Problema D bucket sort

SÃO PAULO

2024

1. Descrição do problema

O *Bucket sort* é um algoritmo de ordenação, cuja estratégia para ordenar é dividir o vetor em “buckets” (baldes) com um número finito de elementos. A vantagem dos baldes é a sua complexidade O(n), onde “n” é o tamanho do vetor. Entretanto, a complexidade linear é atingida apenas quando os números estão distribuídos uniformemente.

A entrada consiste em duas partes: os parâmetros e o conteúdo que será ordenado. Os primeiros quatro números são a quantidade de caracteres do valores, o tamanho do vetor, o offset e o número de buckets. O restante dos parâmetros são os números inteiros para serem ordenados.

A saída consiste nos números ordenados.

Input example	Output example
5 100000 48 9 99997 99996 99995 99994 99993 99992 99991 99990 99989 99988 99987 99986 ...	00000 00001 00002 00003 00004 00005 00006 00007 00008 00009 00010 00011 ...

2. Solução serial

2.1 Fluxo

O fluxo do código pode ser dividido em quatro partes: carregamento dos dados, distribuição dos dados em cada balde, ordenação dos baldes e exibição dos resultados.

O carregamento dos dados é a primeira etapa do programa, os dados incluem os parâmetros especificados no enunciado do problema e uma sequência de números inteiros que serão alocados em um vetor para serem ordenados. Para manipular os números, a estratégia usada foi separar um espaço na memória para cada elemento e trabalhar com ponteiros para obter o índice do primeiro caractere de cada número, para isto, os números são tratados como um vetor de caracteres.

```
void load(char **vString, long int *N, int *l, int *os, int *nb){
  char *strings = NULL;
  int     len,
          offset,
          nbucket;
  long int n;
```

```
  if (!fscanf(stdin, "%d", &len)) {
    printf ("ERROR len\n");
    exit (1);
  }

  if (!fscanf(stdin, "%ld", &n)) {
    printf ("ERROR n\n");
    exit (1);
  }

  if (!fscanf(stdin, "%d", &offset)) {
    printf ("ERROR offset\n");
    exit (1);
  }

  if (!fscanf(stdin, "%d", &nbucket)) {
    printf ("ERROR nbucket\n");
    exit (1);
  }

  nbucket = nbucket + 1;
  len = len + 1;

  strings = (char*) malloc(n * len);

  for (int i = 0; i < n; i++){
    if (!fscanf(stdin, "%s", strings + (i * len))) {
    printf ("ERROR %d\n",i);
    exit (1);
    }
  }

  *vString = strings;
  *N = n;
  *l = len;
  *os = offset;
  *nb = nbucket;
}
```

Com os dados já carregados, a próxima etapa é criar os baldes e distribuir os elementos do vetor entre eles. O número de baldes é um dos parâmetros pré-definidos na entrada do problema e, para cada balde criado, o índice do primeiro caractere de cada número é armazenado como referência, juntamente com o tamanho de cada balde.

```
long int* bucket_sort(char *a, int length, long int size, int offset, int nbuckets) {

   long int i;
   bucket   *buckets = NULL,
            *b = NULL;
   long int *returns = NULL;

   returns = (long int *) malloc(sizeof(long int) * size);
   buckets = (bucket   *) malloc(sizeof(bucket) * nbuckets);


   for (i = 0; i < nbuckets; i++) {
      buckets[i].data = returns + i * size / nbuckets;
      buckets[i].length = length;
      buckets[i].total = 0;
   }

   for (i = 0; i < size; i++) {
      b = &buckets[*(a + i * length) - offset];
      b->data[b->total++] = i;
   }

   for (i = 0; i < nbuckets; i++)
      sort(a, &buckets[i]);

   return returns;
}
```

Em seguida, é preciso ordenar cada balde; para isto, é preciso usar um algoritmo de ordenação auxiliar. No código original, o algoritmo auxiliar é o *Insertion Sort*, que se baseia em comparações para colocar cada elemento em sua respectiva posição.

Neste caso, comparamos os elementos lexicograficamente e de forma sequencial:

```
void sort(char *a, bucket *bucket) {
   int j, i, length;
   long int key;
   length = bucket->length;

   for (j = 1; j < bucket->total; j++) {
      key = bucket->data[j];
      i = j - 1;
      while (i >= 0 && strcmp(a + bucket->data[i] * length, a + key * length) > 0) {
         bucket->data[i + 1] = bucket->data[i];
         i--;
      }
      bucket->data[i + 1] = key;
   }
}
```

A última etapa é a de exibição dos resultados. Originalmente o vetor já ordenado era exibido na saída padrão, mas essa parte foi alterada para salvar os resultados no arquivo *output.txt*, permitindo uma melhor visualização dos dados.

```
void save(char *strings, long int *index, long int n, int len){
  for (int i = 0; i < n; i++){
    printf("%s\n", strings + (index[i] * len));
  }
}
```

Antes de finalizar o código, todas as memórias alocadas são liberadas.

2.2 Oportunidades de paralelismo

Durante a análise do código sequencial, o grupo identificou que duas das quatro partes são inerentemente sequências, o carregamento dos dados e salvar os resultados. As outras duas partes, no entanto, poderiam ser paralelizadas para melhorar o desempenho.

2.2.1 Ordenação dos baldes

Na abordagem tradicional, a ordenação dos baldes no *Bucket Sort* é realizada em um laço de repetição de forma sequencial, onde cada balde é ordenado individualmente, um após o outro. Esta etapa pode ser paralelizada, sendo o grão um laço de repetição, para melhorar o desempenho.

Com este objetivo, foi utilizada a anotação “parallel for” da biblioteca *OpenMP*, que cria uma região paralela para executar as iterações do laço de forma simultânea. Permitindo que cada balde seja ordenado de forma independente e aproveitando ao máximo o uso dos recursos computacionais.

```
#pragma omp parallel for shared(a, buckets)
for (i = 0; i < nbuckets; i++)
  sort(a, &buckets[i]);
```

Outras abordagens foram consideradas para paralelizar a ordenação dos baldes, como o uso de um saco de tarefas, onde cada balde seria uma tarefa. Porém, estas abordagens se mostraram menos eficientes em testes que envolviam outros algoritmos de ordenação nos baldes.

2.2.2 Algoritmo de ordenação auxiliar

O *Insertion Sort*, usado como algoritmo auxiliar do *Bucket Sort* no código base, possui complexidade $O(n^2)$ e, embora teoricamente seja possível paralelizá-lo, duas abordagens principais apresentam limitações práticas:

1. Divisão em sub vetores menores: consiste em dividir o vetor em partes menores e ordená-las separadamente. Isto pode ser feito utilizando o *Merge* (típico do *Merge Sort*) para juntar os sub vetores ordenados ou o *Partition* (típico do *Quick Sort*) para auxiliar na separação dos sub vetores. No entanto, esta abordagem foi descartada porque seria necessário introduzir um terceiro algoritmo no código do *Bucket Sort*, adicionando uma complexidade desnecessária.
2. Busca paralela pela posição do elemento: se baseia em utilizar a busca (binária ou linear) pela posição do elemento no vetor de forma paralela. Contudo, após determinar esta posição, seria necessário percorrer a extensão do vetor ajustando a posição dos demais elementos, processo que não pode ser paralelizado.

Por estas razões, a ideia de paralelizar o *Insertion Sort* foi descartada, abrindo espaço para o uso de outros algoritmos de ordenação que possuem melhor desempenho quando paralelizados (3.0).

3. Análise dos algoritmos de ordenação

Na busca pelo algoritmo de ordenação que apresenta maior eficiência no cenário retratado, são explorados os quatro algoritmos clássicos: *Odd Even Sort*, *Merge Sort*, *Quick Sort* e *Radix Sort*. Estes algoritmos foram escolhidos por sua eficiência e pela possibilidade de serem paralelizados.

Os testes de cada algoritmo foram realizados 10 vezes em condições controladas, utilizando um ambiente com 4 CPUs e 8 GB de memória RAM, onde todos os recursos computacionais foram dedicados exclusivamente para a execução dos algoritmos de ordenação. Cada algoritmo também foi testado variando o número de processadores disponíveis em 1, 2, 3 e 4 cores com a variável de ambiente *OMP_NUM_THREADS* da biblioteca *OpenMP*.

A entrada utilizada em cada teste também foi a mesma, composta por um vetor de 100.000 dados.

Para uma análise abrangente e justa entre os algoritmos foram utilizadas as métricas:

1. Média: reflete o tempo médio de execução do algoritmo durante os testes, fornecendo uma visão geral do algoritmo.
2. Mediana: apresenta o valor de execução central, eliminando possíveis distorções.

3. Desvio padrão: mede a consistência do desempenho, indicando o grau de variação entre os tempos de execução.
4. Speedup: avalia a eficiência da paralelização, comparando a velocidade da versão paralela com a versão sequencial.

Essas métricas permitem uma análise detalhada do comportamento de cada algoritmo, em termos de tempo de execução, escalabilidade e eficiência.

Todavia, após os primeiros testes, os números de Speedup estavam “estranhos” nos algoritmos utilizando o *Merge Sort* e o *Radix Sort*. Isso porque os algoritmos estavam com tempos rápidos, mas a versão paralela estava executando em um tempo quase igual ao tempo da versão sequencial.

Visando entender o que estava acontecendo, foram feitas algumas análises seguindo a Lei de Amdahl, responsável pela análise como uma mesma entrada de dados pode variar com relação ao número de processadores.

3.1 Algoritmo original

O algoritmo original do *Bucket Sort* está usando o *Insertion Sort* como auxiliar para a ordenação dos baldes. As métricas obtidas ao executar o programa original foram:

Algoritmo sequencial	
Execução	Tempo (s)
1	204,897
2	204,323
3	208,977
4	198,082
5	202,422
6	201,020
7	200,976
8	195,338
9	198,322
10	201,157

Tabela 1: dados de tempo de execução do algoritmo sequencial.

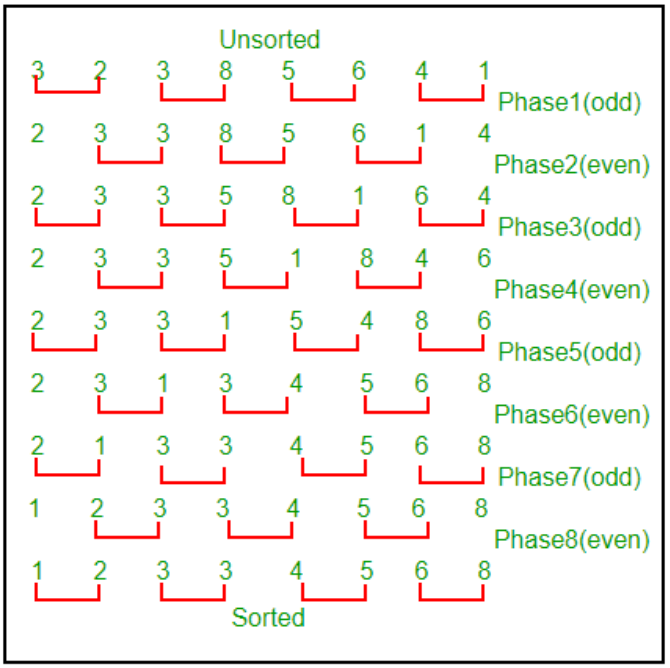
3.2 Odd-Even Sort[1]

O algoritmo *Odd-Even Sort* é uma variação do algoritmo do *Bubble Sort*, um dos algoritmos de ordenação mais simples. O *Bubble Sort* tem seu funcionamento baseado na comparação dos pares adjacentes de um vetor e na troca de posições quando necessário, visando "empurrar" o menor elemento gradativamente para o início do vetor. Apesar de ser um algoritmo simples, possui complexidade $O(n^2)$, não sendo eficiente para grandes conjuntos de dados; além disso, as trocas e comparações dificultam na sua paralelização, pois cada iteração depende diretamente dos resultados obtidos na iteração anterior.

O *Odd-Even Sort* tem como objetivo quebrar as dependências entre cada iteração, facilitando a sua paralelização. Dessa forma, ao invés de percorrer todos os elementos do vetor em uma única "passada" para as comparações, o algoritmo realiza duas etapas separadas:

1. Comparação e, se necessário, troca dos elementos de índice ímpar (odd) e seus sucessores.
2. Comparação e, se necessário, troca dos elementos de índice par (even) e seus sucessores.

Este processo alternado permite múltiplas comparações independentes que podem ser executadas de forma paralela. Isso porque, cada comparação na "passada ímpar" é independente das outras, o mesmo ocorre na "passada par". Essa independência fica evidente no esquema abaixo:



[1] Sah, "Parallelizing Sorting Algorithms using OpenMP".

Figura 1: representação das fases do algoritmo Odd-Even.[2]

Dessa forma, o grão da paralelização seria o laço de repetição responsável por paralelizar cada passada.

3.2.1 Algoritmo sequencial

```
void odd_even(char *a, long int *data, int length, int n) {
      int sorted = 0;

      while(!sorted) {
        sorted = 1;

        // Odd
        for(int i = 1; i < n - 1; i += 2) {
          if(strcmp(a + (data[i] * length), a + (data[i + 1] * length)) > 0) {
              long int temp = data[i];
              data[i] = data[i + 1];
              data[i + 1] = temp;
              sorted = 0;
          }
        }

        // Even
        for(int i = 0; i < n - 1; i += 2) {
          if(strcmp(a + (data[i] * length), a + (data[i + 1] * length)) > 0) {
              long int temp = data[i];
              data[i] = data[i + 1];
              data[i + 1] = temp;
              sorted = 0;
          }
        }
      }
}
```

3.2.2 Algoritmo paralelo[3]

```
void odd_even(char *a, long int *data, int length, int n) {
  int sorted = 0;

  while(!sorted) {
    sorted = 1;

    #pragma omp parallel for shared(a, data, n, length, sorted) default(none)
    // Odd
```

[2] Sah.
[3] Sah.

```
    for(int i = 1; i < n - 1; i += 2) {
      if(strcmp(a + (data[i] * length), a + (data[i + 1] * length)) > 0) {
        long int temp = data[i];
        data[i] = data[i + 1];
        data[i + 1] = temp;
        sorted = 0;
      }
    }

    #pragma omp parallel for shared(a, data, n, length, sorted) default(none)
    // Even
    for(int i = 0; i < n - 1; i += 2) {
      if(strcmp(a + (data[i] * length), a + (data[i + 1] * length)) > 0) {
        long int temp = data[i];
        data[i] = data[i + 1];
        data[i + 1] = temp;
        sorted = 0;
      }
    }
  }
}
```

3.2.3 Métricas

A análise dos testes do algoritmo *Odd-Even Sort* paralelo revela um ganho expressivo de desempenho em comparação com a versão sequencial. Essa melhoria é evidenciada pelas métricas de tempo de execução coletadas:

Odd Even Sort									
Algoritmo sequencial		Algoritmo paralelo (1 core)		Algoritmo paralelo (2 cores)		Algoritmo paralelo (3 cores)		Algoritmo paralelo (4 cores)	
Execução	Tempo	Execução	Tempo	Execução	Tempo	Execução	Tempo	Execução	Tempo
1	221,520	1	250,082	1	243,781	1	228,911	1	72,803
2	223,224	2	260,203	2	176,772	2	235,828	2	71,800
3	219,759	3	228,189	3	200,246	3	213,171	3	73,489
4	225,215	4	230,236	4	177,285	4	212,736	4	71,729
5	222,565	5	237,736	5	177,637	5	250,646	5	71,605
6	224,860	6	244,079	6	205,832	6	250,646	6	71,908
7	223,824	7	227,934	7	204,860	7	247,636	7	71,905
8	230,098	8	225,033	8	229,947	8	234,834	8	71,568
9	236,048	9	226,606	9	181,947	9	272,548	9	72,198
10	228,082	10	240,029	10	181,721	10	283,162	10	72,166

Tabela 2: dados de tempo de execução do *Odd-Even Sort*.

Odd Even Sort									
Algoritmo sequencial		Algoritmo paralelo (1 core)		Algoritmo paralelo (2 cores)		Algoritmo paralelo (3 cores)		Algoritmo paralelo (4 cores)	
Média	225,520	Média	237,013	Média	198,003	Média	243,012	Média	72,117
Mediana	224,342	Mediana	233,986	Mediana	191,097	Mediana	241,732	Mediana	71,907
Desvio Padrão	4,527234393	Desvio Padrão	11,07324341	Desvio Padrão	22,4656419	Desvio Padrão	21,7871161	Desvio Padrão	0,5714144643

Tabela 3: métricas do *Odd-Even Sort*.

Os resultados indicam que, para 1 e 3 cores, o algoritmo paralelo não apresentou grande eficiência, o que pode ser atribuído à forma como as tarefas são distribuídas pelo escalonador. No entanto, ao utilizar 2 cores, o algoritmo mostrou uma melhora significativa em relação ao código sequencial, que foi cerca de 30 segundos mais lento. E as execuções com 4 processadores se destacaram, com uma redução de três vezes no tempo total de execução em comparação ao código sequencial.

Ao analisar os tempos de execução das diferentes partes do código, observa-se que a porção que pode ser paralelizada, responsável pela ordenação, corresponde a aproximadamente 99% do tempo total de execução do algoritmo. Essa informação é crucial, pois possibilita a implementação da Lei de Amdahl para estimar o ganho de desempenho à medida que o número de processadores aumenta.

Odd Even Sort				
Execução	Carregamento	Ordenação	Saída	% paralelizável
1	0,050506	208,840107	0,061047	0,9994661301
2	0,048604	211,932168	0,066675	0,9994563528
3	0,066675	209,005305	0,076723	0,999314373
4	0,049614	208,384901	0,071159	0,9994207688
5	0,052868	211,017792	0,076421	0,9993876827
6	0,048122	213,023183	0,078211	0,9994073034
7	0,060124	214,810552	0,070812	0,9993908296
8	0,057143	216,173419	0,065333	0,9994337572
9	0,050159	215,499776	0,069776	0,9994437661
10	0,055208	216,001365	0,070207	0,9994197156

Tabela 4: dados de tempo de execução separando a parte paralela do *Odd-Even Sort*.

Odd Even Sort		
Processadores	Amdahl	Obtido
1	1	0,9515080837
2	1,998841156	1,138971267
3	2,996525482	0,9280187217
4	3,993054985	3,127129349
8	7,967664788	-
16	15,86205772	-

Tabela 5: dados teóricos do Speedup do *Odd-Even Sort*.

Com base nos dados teóricos derivados da Lei de Amdahl, os valores de Speedup para 2 e 4 processadores são elevados, indicando que o desempenho do algoritmo está se aproximando de seu limite teórico de paralelização. No entanto, os valores de Speedup para 1 e 3 processadores sugerem que há potencial para uma utilização mais eficiente dos recursos computacionais, indicando que o algoritmo ainda pode possuir partes que podem ser paralelizadas para melhorar o desempenho.

A partir dos valores de Speedup, é possível calcular a eficiência do algoritmo, que reflete a porcentagem de uso de cada processador durante as execuções.

Odd Even Sort	
Processadores	Eficiência
1	0,9515080837
2	0,5694856335
3	0,3093395739
4	0,7817823373

Tabela 6: dados de eficiência do *Odd-Even Sort*.

Os valores de eficiência indicam que as execuções com 2 e 3 processadores não fizeram um uso otimizado dos recursos disponíveis, especialmente com 3 processadores, onde a eficiência foi mais limitada. No entanto, com 4 processadores, a eficiência quase atinge 80%, o que demonstra que nessas condições, o algoritmo utiliza melhor os recursos disponíveis.

Para entender melhor o comportamento do algoritmo perante a uma variação de processadores, é preciso realizar testes mais extensos e com uma variação maior na quantidade de processadores.

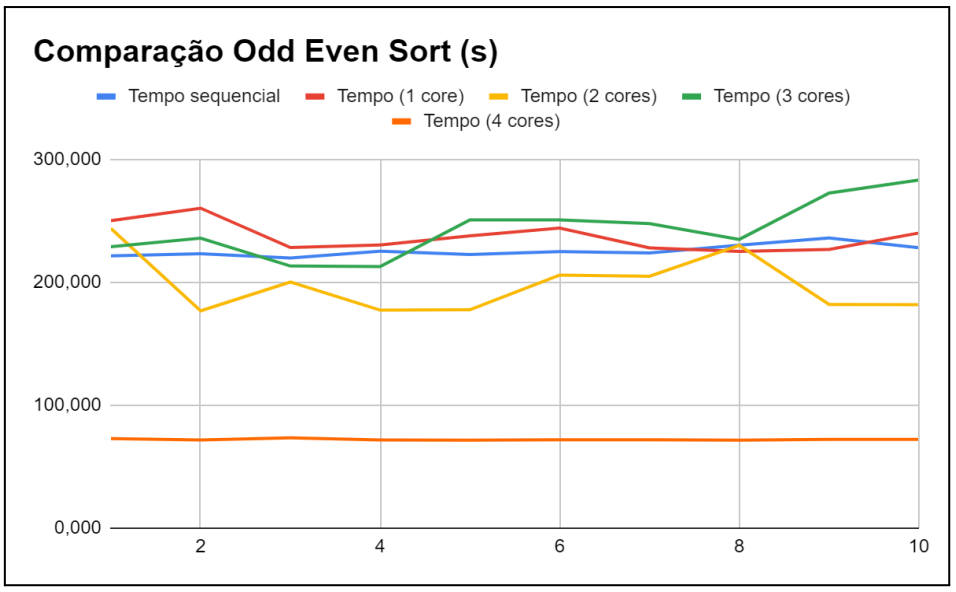


Figura 2: gráfico do desempenho do algoritmo *Odd-Even*.

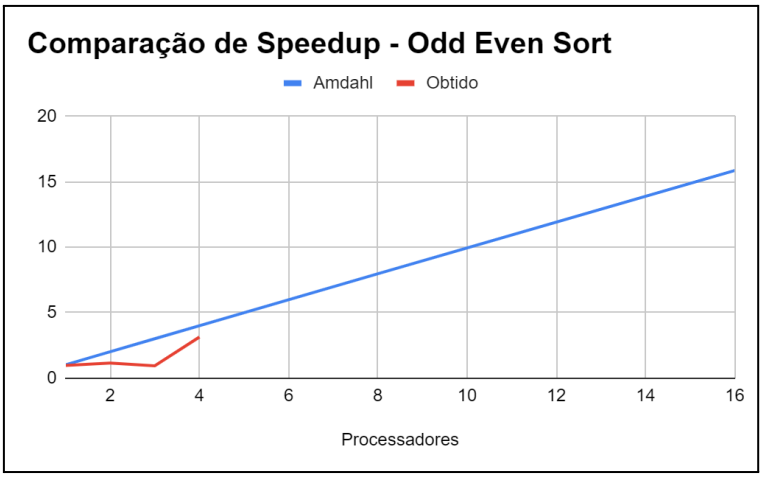


Figura 3: gráfico do cálculo de Speedup do *Odd-Even*.

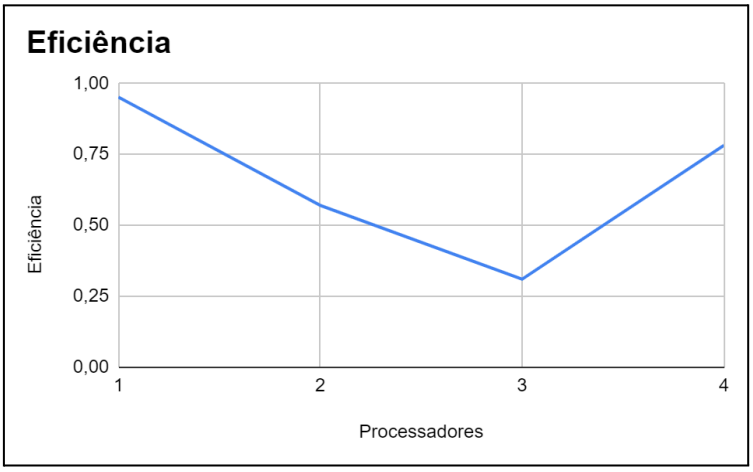


Figura 4: gráfico do cálculo de eficiência do *Odd-Even*.

3.3 Merge sort[4]

O *Merge Sort* é um algoritmo de ordenação eficiente, de complexidade O(n log n), que utiliza a estratégia de divisão e conquista. Sua abordagem é dividir o vetor recursivamente em subvetores menores até que cada subvetor contenha apenas um elemento que, por definição, está ordenado. Em seguida, os subvetores são intercalados de forma ordenada.

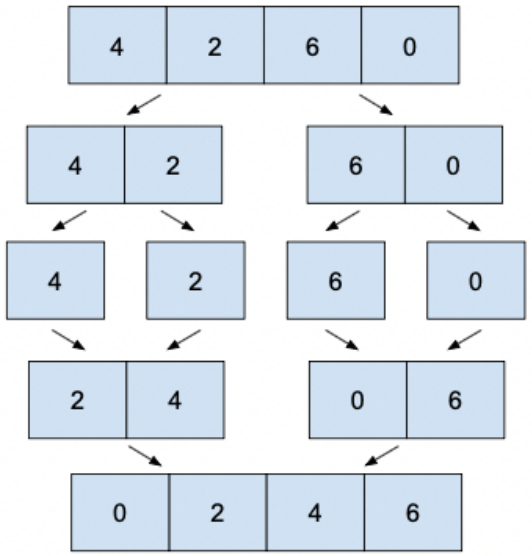


Figura 5: representação da recursão no *Merge Sort*[5]

Para intercalar os elementos do vetor, utiliza-se uma função chamada *Merge*, que recebe duas listas ordenadas e combina em uma única lista também ordenada. O algoritmo *Merge* que foi utilizado no código do *Bucket Sort* é o mais tradicional:

```
void merge(char *a, long int *data, int left, int middle, int right, int length) {
   long int n1 = middle - left + 1;
   long int n2 = right - middle;

   long int *L = (long int *) malloc(n1 * sizeof(long int));
   long int *R = (long int *) malloc(n2 * sizeof(long int));

   for (int i = 0; i < n1; i++) {
      L[i] = data[left + i];
   }

   for (int j = 0; j < n2; j++) {
      R[j] = data[middle + 1 + j];
   }

   long int i = 0, j = 0, k = left;
```

[4] Sah.
[5] stevenard, “Merge Sort”.

```
    while (i < n1 && j < n2) {
        if (strcmp(a + (L[i] * length), a + (R[j] * length)) < 0) {
            data[k] = L[i];
            i++;
        } else {
            data[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        data[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        data[k] = R[j];
        j++;
        k++;
    }

    free(L);
    free(R);
}
```

O Merge é um algoritmo sequencial; entretanto, é possível paralelizar o *Merge Sort* aproveitando a independência dos subvetores em cada nível da árvore de recursão, sendo cada subvetor o grão da paralelização no nível de tarefas.

Uma abordagem eficiente para paralelizar o *Merge Sort* é realizar subdivisões de forma paralela até uma determinada profundidade da árvore, permitindo que os subvetores gerados sejam processados simultaneamente. Para a ordenação simultânea, cada subvetor é colocado dentro de uma tarefa e, para prosseguir para o Merge, a ordenação de cada vetor precisa ser concluída.

No caso, a profundidade escolhida foi 3, o que resulta na divisão do vetor original em 8 subvetores que serão ordenados sequencialmente. Essa profundidade foi definida após testes rápidos, considerando o tempo limitado para a experimentação.

Outra abordagem possível para restringir o paralelismo é limitar pela quantidade de elementos no subvetor, mas esta estratégia demandaria mais testes aprofundados para identificar o ponto de paralelismo. Além disso, é preciso levar em conta que a melhor

estratégia depende da arquitetura e dos recursos computacionais utilizados, não havendo uma resposta “certa”.

3.3.1 Algoritmo sequencial

```
void mergesort(char *a, long int *data, int left, int right, int length) {
   if(left < right) {
      int middle = (left + right) / 2;

      mergesort(a, data, left, middle, length);
      mergesort(a, data, middle + 1, right, length);

      merge(a, data, left, middle, right, length);
   }
}
```

3.3.2 Algoritmo paralelo

```
void mergesort(char *a, long int *data, int left, int right, int length, int depth) {
   if(left < right) {
      int middle = (left + right) / 2;

      if(depth < 3) {
         #pragma omp task shared(a, data, length)
         mergesort(a, data, left, middle, length, depth + 1);
         #pragma omp task shared(a, data, length)
         mergesort(a, data, middle + 1, right, length, depth + 1);
      } else {
         mergesort(a, data, left, middle, length, depth + 1);
         mergesort(a, data, middle + 1, right, length, depth + 1);
      }

      #pragma omp taskwait
      merge(a, data, left, middle, right, length);
   }
}
```

3.3.3 Métricas

A análise das métricas obtidas a partir dos testes realizados com a versão paralela do *Merge Sort* indica que a solução não apresentou ganhos significativos de eficiência. Esse comportamento é evidenciado pelas tabelas a seguir:

Merge Sort									
Algoritmo sequencial		Algoritmo paralelo (1 core)		Algoritmo paralelo (2 cores)		Algoritmo paralelo (3 cores)		Algoritmo paralelo (4 cores)	
Execução	Tempo (s)	Execução	Tempo (s)	Execução	Tempo (s)	Execução	Tempo (s)	Execução	Tempo (s)
1	0,195	1	0,241	1	0,236	1	0,252	1	0,205
2	0,237	2	0,310	2	0,274	2	0,229	2	0,254
3	0,246	3	0,282	3	0,230	3	0,247	3	0,229
4	0,215	4	0,227	4	0,191	4	0,257	4	0,215
5	0,225	5	0,281	5	0,266	5	0,216	5	0,211
6	0,233	6	0,268	6	0,219	6	0,302	6	0,211
7	0,236	7	0,320	7	0,261	7	0,270	7	0,208
8	0,239	8	0,253	8	0,323	8	0,225	8	0,230
9	0,306	9	0,249	9	0,290	9	0,229	9	0,210
10	0,257	10	0,227	10	0,381	10	0,358	10	0,260

Tabela 7: dados de tempo de execução do *Merge Sort*.

Merge Sort									
Algoritmo sequencial		Algoritmo paralelo (1 core)		Algoritmo paralelo (2 cores)		Algoritmo paralelo (3 cores)		Algoritmo paralelo (4 cores)	
Média	0,239	Média	0,266	Média	0,267	Média	0,259	Média	0,223
Mediana	0,237	Mediana	0,261	Mediana	0,264	Mediana	0,250	Mediana	0,213
Desvio Padrão	0,027602 35497	Desvio Padrão	0,030759 71391	Desvio Padrão	0,052112 28262	Desvio Padrão	0,040951 80094	Desvio Padrão	0,018665 74402

Tabela 8: métricas do *Merge Sort*.

O desempenho limitado do *Merge Sort* nas execuções com um número menor de processadores foi relacionado ao caráter altamente paralelo dos baldes e da ordenação. Esse tipo de paralelismo dentro de outro paralelismo pode acabar sobrecarregando os processadores do ambiente de teste, o que prejudica o desempenho geral.

Esse comportamento é refletido com Speedups abaixo de 1 para as execuções com 1, 2 e 3 processadores. Além disso, o Speedup de 1,07 com 4 processadores sugere apenas um leve aumento de eficiência da versão paralela em comparação com a versão sequencial.

Analisando com mais afinco a falta de eficiência, principalmente nas execuções com 4 processadores, o grupo separou a parte do código que precisa ser executada e parte passível de paralelização. A partir desta análise, foi possível observar que apenas 38,5% do tempo total de execução está sendo utilizado na parte paralela. Isso indica que a porção paralela do algoritmo é relativamente curta em relação ao tempo total de execução, o que, por si só, não é suficiente para gerar um ganho significativo de desempenho.

Merge Sort				
Execução	Carregamento	Ordenação	Saída	% paralelizável
1	0,051741	0,076326	0,062866	0,3997527929
2	0,055991	0,081531	0,076038	0,3817709309
3	0,050171	0,07337	0,065075	0,3889913899
4	0,049501	0,077732	0,077732	0,379245237
5	0,049722	0,074826	0,066345	0,3919787525
6	0,050162	0,074253	0,069888	0,3821505587
7	0,05114	0,079765	0,064718	0,4077485776
8	0,068393	0,08178	0,06776	0,3752529447
9	0,049377	0,073881	0,064865	0,3927270988
10	0,052064	0,077633	0,07516	0,37896191

Tabela 9: dados de tempo de execução separando a parte paralela do *Merge Sort*.

Essa conclusão é reforçada pela aplicação da Lei de Amdahl, que estima o Speedup máximo de um sistema baseado na proporção de código paralelizável. Ao aplicar essa lei aos tempos de execução das partes sequenciais e paralelas, observa-se que os valores de Speedup não apresentam variações significativas e são próximos dos valores obtidos.

Merge Sort		
Processadores	Amdahl	Obtido
1	1	0,8987960873
2	1,238828074	0,894421565
3	1,345980736	0,9241779497
4	1,406822418	1,069861173
8	1,509148312	-
16	1,566103968	-

Tabela 10: dados teóricos do Speedup do *Merge Sort*.

Os valores de Speedup teóricos e obtidos através dos testes, mostra que podem existir pontos passíveis de paralelização, mas que não foram tratados. Além disso, o algoritmo poderia ter uma eficiência maior ao ser executado com uma entrada de dados maior. Uma

maneira de aprofundar a análise do desempenho em cenários com maior volume de dados seria utilizar a Lei de Gustafson. Esta lei é mais adequada para avaliar a escalabilidade de algoritmos, já que ela considera a variação proporcional da entrada de dados em relação à quantidade de processadores.

Pelos valores de Speedup, é possível calcular também a eficiência dos algoritmos, para medir a porcentagem de uso de cada processador durante a execução.

Merge Sort	
Processadores	Eficiência
1	0,8987960873
2	0,4472107825
3	0,3080593166
4	0,2674652933

Tabela 11: dados de eficiência do *Merge Sort*.

A eficiência do Merge Sort diminuiu à medida que o número de processadores aumentou. Isso pode ser explicado pelo fato de que a parte paralelizável do algoritmo, que já era relativamente pequena, foi subdividida entre um número maior de processadores, resultando em maior ociosidade.

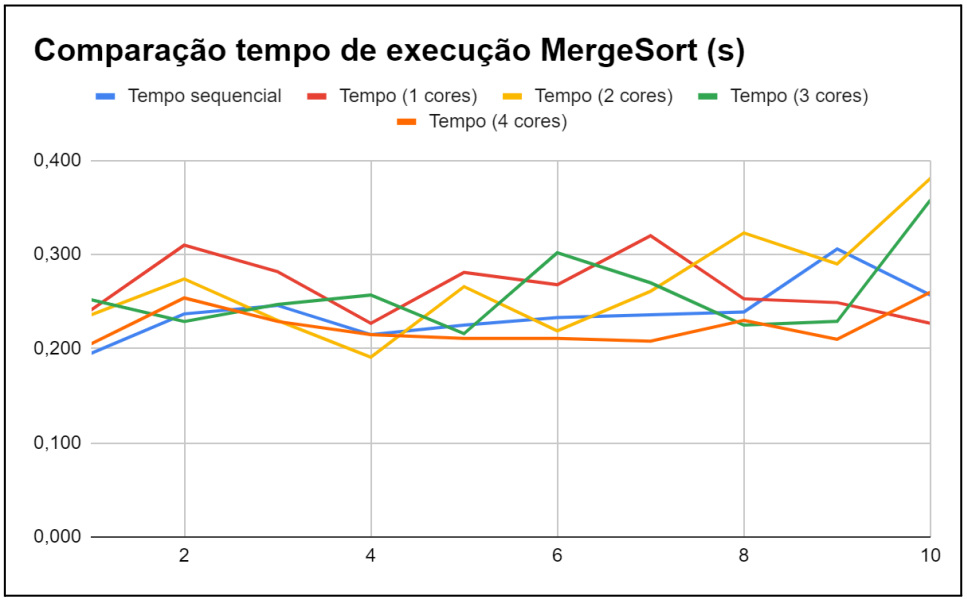


Figura 6: gráfico do desempenho do algoritmo *Merge Sort*.

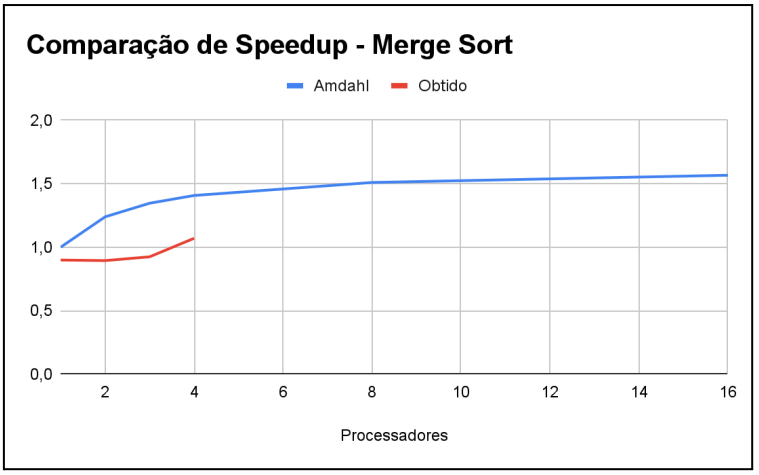


Figura 7: gráfico do cálculo de Speedup do *Merge Sort*.

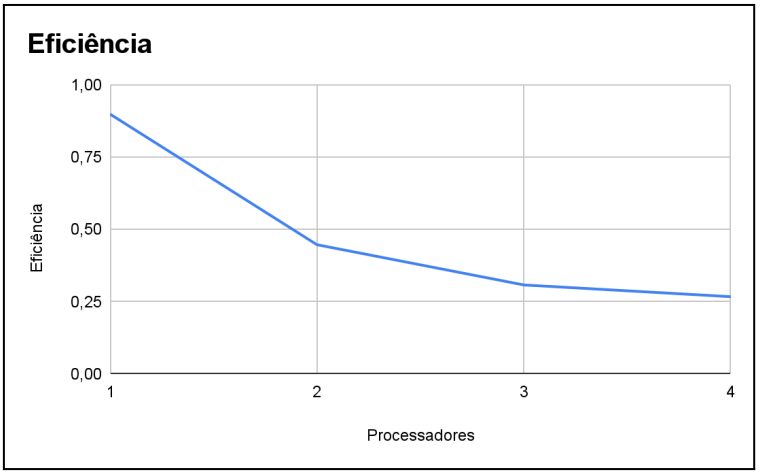


Figura 8: gráfico do cálculo de eficiência do *Merge Sort*.

3.4 Quick sort[6]

Assim como o *Merge Sort*, o *Quick Sort* também utiliza a técnica de divisão e conquista para ordenar um vetor. O *Quick Sort*, em geral, é um algoritmo eficiente, com complexidade O(n log n) no seu caso médio; no entanto, em seu pior caso a complexidade aumenta significativamente, passando a ser $O(n^2)$.

A divisão do vetor no *Quick Sort*, é realizada por uma função chamada *Partition*, onde um pivô é escolhido e todos os elementos menores que o pivô são rearranjados para o início do vetor e os elementos maiores são rearranjados para o final. Após todos os elementos serem separados entre menores e maiores que o pivô, o pivô já está ordenado na sua posição correta e o processo é repetido recursivamente nos subvetores que se formaram à esquerda e à direita do pivô, até que todo o vetor esteja ordenado.

[6] "Ordenação: Quicksort".

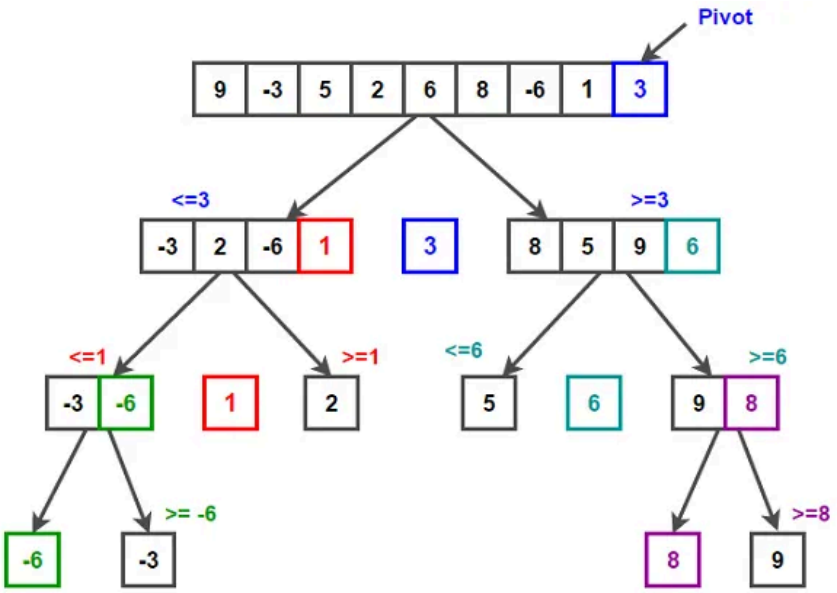


Figura 9: representação da recursão no *Quick Sort*[7]

Dentro do *Partition*, existem várias formas de escolher um pivô, a abordagem utilizada foi a de escolher o primeiro elemento do vetor.

```
int partition(char *a, long int *data, int low, int high, int length) {
   long int pivot = data[high];
   int i = low - 1;

   for (int j = low; j < high; j++) {
      if (strcmp(a + data[j] * length, a + pivot * length) <= 0) {
         i++;
         long int temp = data[i];
         data[i] = data[j];
         data[j] = temp;
      }
   }

   long int temp = data[i + 1];
   data[i + 1] = data[high];
   data[high] = temp;

   return i + 1;
}
```

É possível paralelizar o *Quick Sort* modificando o processo de *Partition* para selecionar vários pivôs, dividindo o vetor em vários subvetores para ordená-los simultaneamente. Entretanto, essa abordagem pode acarretar um custo computacional significativo, principalmente ao selecionar mais de 3 pivôs. Teoricamente, essa abordagem é promissora, mas exige mais tempo para testes de eficiência e otimização do algoritmo.

[7] "Ordenação: Quicksort".

Dessa forma, a abordagem adotada foi simplificar o paralelismo, limitando a árvore de recursão até a profundidade 3 (semelhante ao que foi feito no *Merge Sort*). Neste ponto, o vetor original é dividido em 8 subvetores e esses subvetores são ordenados de forma sequencial, mas simultânea, onde cada um é designado a uma tarefa (grão do paralelismo).

Esta abordagem reduz o custo computacional, uma vez que o paralelismo é aplicado apenas até uma profundidade limitada da recursão. A profundidade 3 foi escolhida mediante a testes rápidos, mas que poderiam ser aprofundados para identificar o momento ideal de paralelismo.

3.4.1 Algoritmo sequencial

```
void quicksort(char *a, long int *data, int low, int high, int length) {
   if (low < high) {
      int pi = partition(a, data, low, high, length);

      quicksort(a, data, low, pi - 1, length);
      quicksort(a, data, pi + 1, high, length);
   }
}
```

3.4.2 Algoritmo paralelo

```
void sort(char *a, bucket *bucket) {
 #pragma omp parallel
 {
  #pragma omp single
  quicksort(a, bucket->data, 0, bucket->total - 1, bucket->length, 0);
 }
}

void quicksort(char *a, long int *data, int low, int high, int length, int depth) {
 if (low < high) {
  int pi = partition(a, data, low, high, length);

  if (depth < 3) {
        #pragma omp task shared(a, data, length) if (high - low > 1000)
    quicksort(a, data, low, pi - 1, length, depth + 1);

        #pragma omp task shared(a, data, length) if (high - low > 1000)
    quicksort(a, data, pi + 1, high, length, depth + 1);

        #pragma omp taskwait
  } else {
    quicksort(a, data, low, pi - 1, length, depth + 1);
```

```
    quicksort(a, data, pi + 1, high, length, depth + 1);
   }
  }
}
```

3.4.3 Métricas

A análise das métricas revela que a versão sequencial do algoritmo *Quick Sort* teve um desempenho inferior do que o esperado para um algoritmo com complexidade O(n log n). Este comportamento foi causado pela entrada de dados que está sendo utilizada para teste, que está ordenada ao contrário, representando o pior caso do *Quick Sort* e aumentando a complexidade do algoritmo para $O(n^2)$.

Quick Sort									
Algoritmo sequencial		Algoritmo paralelo (1 core)		Algoritmo paralelo (2 cores)		Algoritmo paralelo (3 cores)		Algoritmo paralelo (4 cores)	
Execução	Tempo	Execução	Tempo (s)	Execução	Tempo (s)	Execução	Tempo (s)	Execução	Tempo (s)
1	207,224	1	218,825	1	120,865	1	89,464	1	71,508
2	213,252	2	247,908	2	113,812	2	98,979	2	73,179
3	210,428	3	220,399	3	112,843	3	85,483	3	73,742
4	215,791	4	218,823	4	109,079	4	94,423	4	72,120
5	209,843	5	219,299	5	110,549	5	92,185	5	72,417
6	212,355	6	223,290	6	111,827	6	81,966	6	73,097
7	215,687	7	207,256	7	110,559	7	87,497	7	65,219
8	218,934	8	215,548	8	108,069	8	95,381	8	65,038
9	213,786	9	219,130	9	109,683	9	86,758	9	64,453
10	212,958	10	235,130	10	114,504	10	87,225	10	64,150

Tabela 12: dados de tempo de execução do *Quick Sort*.

Quick Sort									
Algoritmo sequencial		Algoritmo paralelo (1 core)		Algoritmo paralelo (2 cores)		Algoritmo paralelo (3 cores)		Algoritmo paralelo (4 cores)	
Média	213,026	Média	222,561	Média	112,179	Média	89,936	Média	69,492
Mediana	213,105	Mediana	219,215	Mediana	111,193	Mediana	88,481	Mediana	71,814
Desvio Padrão	3,182113 254	Desvio Padrão	10,67437 958	Desvio Padrão	3,494239 917	Desvio Padrão	4,935441 854	Desvio Padrão	3,952263 758

Tabela 13: métricas do *Quick Sort*.

Entretanto, apesar de não ser o resultado esperado, a versão paralela do *Quick Sort* obteve um desempenho significativamente melhor quando comparada à versão sequencial.

Os testes do algoritmo com 1 processador, mostraram uma execução inferior ao tempo sequencial, provavelmente por conta da sobrecarga gerada por um alto grau de paralelismo de funções.

Os demais processadores apresentaram um ganho significativo com relação à versão sequencial, todos reduziram expressivamente o tempo de execução com relação ao código sequencial, demonstrando o impacto positivo na paralelização.

Para entender melhor o comportamento do algoritmo, foi analisada a proporção de tempo dedicada à parte paralelizada em comparação com a parte sequencial. Os resultados mostraram que 99,9% do tempo de execução do *Bucket Sort*, que utiliza o *Quick Sor*t, é dedicado à parte paralela. Isso indica que o algoritmo possui um grande potencial de paralelização, já que a maior parte do trabalho pode ser distribuída entre múltiplos processadores.

Quick Sort				
Execução	Carregamento	Ordenação	Saída	% paralelizável
1	0,048746	206,709555	0,069498	0,9994282974
2	0,050361	211,727691	0,083308	0,9993690732
3	0,057816	209,676416	0,066115	0,9994092908
4	0,049144	219,1597	0,079347	0,9994140542
5	0,057711	209,567081	0,072964	0,9993768412
6	0,072964	209,567081	0,072964	0,9993041538
7	0,0494	205,825667	0,087684	0,9993344233
8	0,087684	206,929982	0,072011	0,9992288606
9	0,050175	215,969094	0,074868	0,9994213494
10	0,054482	211,620057	0,061215	0,9994535784

Tabela 14: dados de tempo de execução separando a parte paralela do *Quick Sort*.

A aplicação da Lei de Amdahl à proporção de 99,9% de paralelização e 0,1% de sequencialidade permite estimar os valores teóricos de Speedup.

Quick Sort		
Processadores	Amdahl	Obtido

1	1	0,9571577744
2	1,998786868	1,898981093
3	2,996362811	2,368635064
4	3,992730029	3,065459051
8	7,966155485	-
16	15,85565	-

Tabela 15: dados teóricos do Speedup do *Quick Sort*.

O valor de Speedup alcançado na implementação do algoritmo paralelo foi próximo ao valor obtido pelos testes realizados. Em todos eles, o Speedup se mostrou elevado e próximo ao limite máximo, com destaque para a execução com 4 processadores, que conseguiu um Speedup de 3,065. Indicando que, sob estas condições, a versão paralela do algoritmo é três vezes mais rápida que a versão sequencial.

Quick Sort	
Processadores	Eficiência
1	0,9571577744
2	0,9494905464
3	0,7895450214
4	0,7663647627

Tabela 16: dados de eficiência do *Quick Sort*.

Os valores de eficiência demonstram um alto grau de uso dos processadores com 1 e 2 processadores. Possuindo um desempenho menor com 3 e 4 processadores, mas ainda sendo um valor elevado.

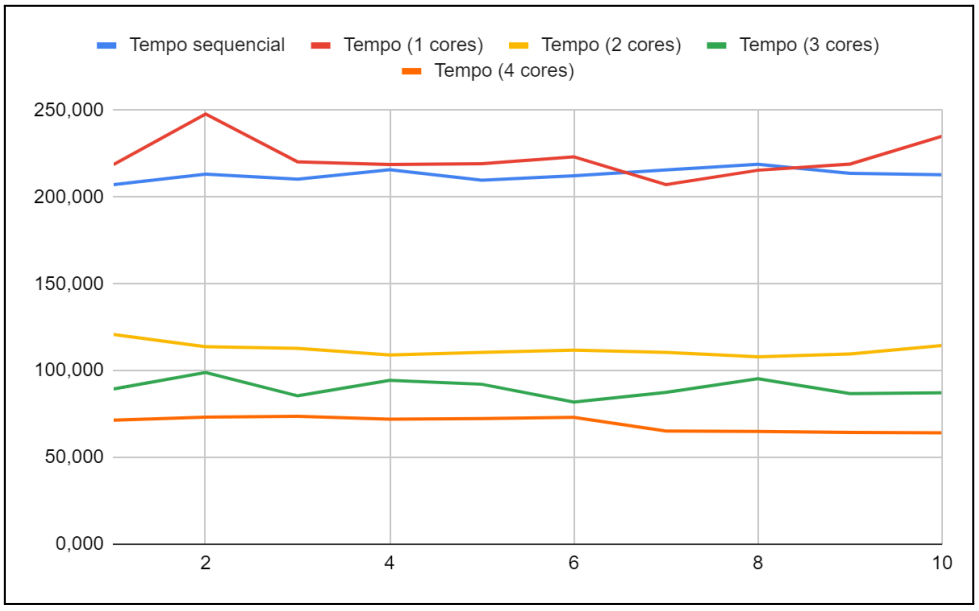


Figura 10: gráfico do desempenho do algoritmo *Quick Sort*.

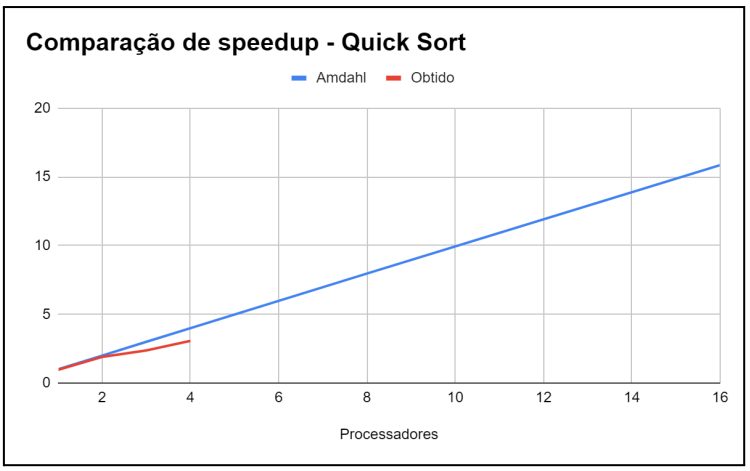


Figura 11: gráfico do cálculo de Speedup do *Quick Sort*.

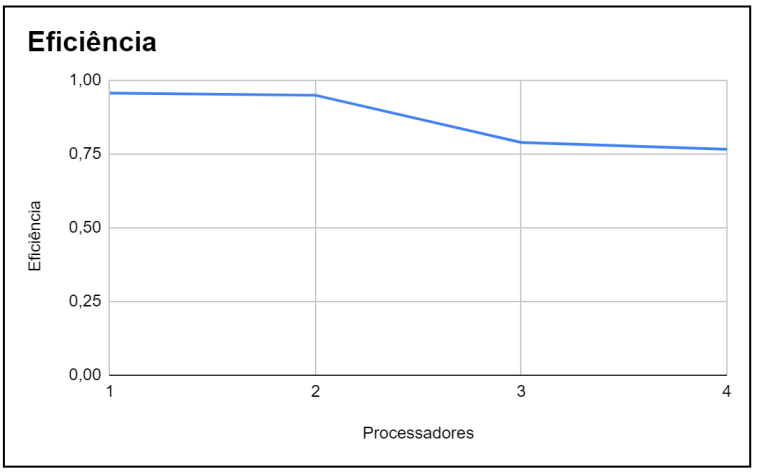


Figura 12: gráfico do cálculo de eficiência do *Quick Sort*.

3.5 Radix sort[8]

Ao contrário dos outros algoritmos de ordenação que foram citados, o *Radix Sort* não se baseia na comparação dos elementos de um vetor para realizar a ordenação. Ao invés disso, ele ordena os elementos com base nos seus dígitos, um de cada vez, começando pelo dígito de maior ordem até o de menor ordem. Isso faz do *Radix Sort* uma excelente escolha para ordenar números inteiros ou conjuntos de caracteres com tamanho fixo, sendo vantajoso no cenário testado.

A ordenação é realizada passo-a-passo com o auxílio de um algoritmo de ordenação estável. No caso, foi utilizado o *counting Sort*, um algoritmo eficiente quando se trata da ordenação de elementos em um intervalo pequeno e com muitas repetições de valores.

[8] "Radix Sort – Data Structures and Algorithms Tutorials".

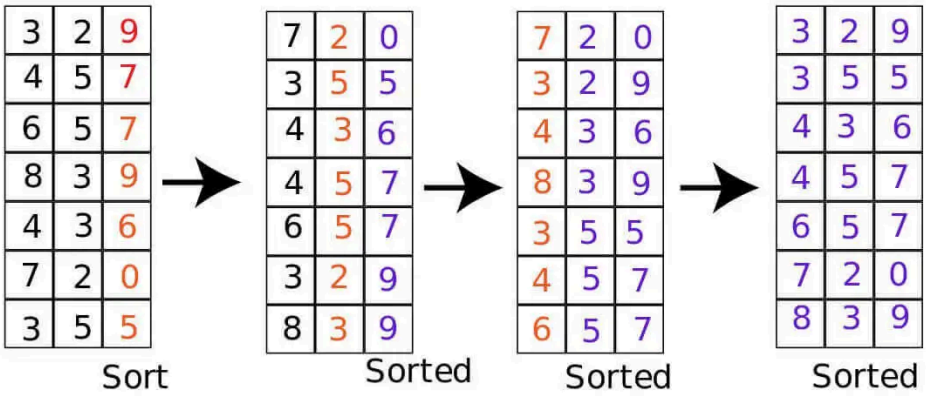


Figura 13: exemplificação do funcionamento do *Radix Sort*.

Dentro deste algoritmo, foram encontrados alguns laços de repetição passíveis de paralelização, como a contagem de ocorrências de cada dígito, a distribuição dos elementos já ordenados e a cópia dos valores ordenados para o vetor original. Estes laços correspondem ao grão de paralelismo do *Radix Sort.*

3.5.1 Algoritmo sequencial

```
void radix(char *a, long int *data, int size, int position, int length) {
   long int *sortedData = (long int *) malloc(size * sizeof(long int));
   int *RADIXCOUNT = (int *) malloc(MAX_RADIX * sizeof(int));

   for (int i = 0; i < MAX_RADIX; i++)
      RADIXCOUNT[i] = 0;

   for (int i = 0; i < size; i++) {
      int valorPos = a[data[i] * length + position];
      RADIXCOUNT[valorPos]++;
   }

   for (int i = 1; i < MAX_RADIX; i++)
      RADIXCOUNT[i] += RADIXCOUNT[i - 1];

   for (int i = size - 1; i >= 0; i--) {
      int valorPos = a[data[i] * length + position];
      sortedData[RADIXCOUNT[valorPos] - 1] = data[i];
      RADIXCOUNT[valorPos]--;
   }

   for (int i = 0; i < size; i++)
      data[i] = sortedData[i];
```

```
    free(sortedData);
    free(RADIXCOUNT);
}

void radixsort(char *a, long int *data, int size, int length) {
    for (int position = length - 1; position >= 0; position--) {
        radix(a, data, size, position, length);
    }
}
```

3.5.2 Algoritmo paralelo

```
void radix(char *a, long int *data, int size, int position, int length) {
    long int *sortedData = (long int *) malloc(size * sizeof(long int));
    int *RADIXCOUNT = (int *) malloc(MAX_RADIX * sizeof(int));

    for (int i = 0; i < MAX_RADIX; i++) {
        RADIXCOUNT[i] = 0;
    }

    #pragma omp parallel for
    for (int i = 0; i < size; i++) {
        int valorPos = a[data[i] * length + position];
        RADIXCOUNT[valorPos]++;
    }

    for (int i = 1; i < MAX_RADIX; i++) {
        RADIXCOUNT[i] += RADIXCOUNT[i - 1];
    }

    #pragma omp parallel for
    for (int i = size - 1; i >= 0; i--) {
        int valorPos = a[data[i] * length + position];
        sortedData[RADIXCOUNT[valorPos] - 1] = data[i];
        RADIXCOUNT[valorPos]--;
    }

    #pragma omp parallel for
    for (int i = 0; i < size; i++) {
        data[i] = sortedData[i];
    }

    free(sortedData);
    free(RADIXCOUNT);
}

void radixsort(char *a, long int *data, int size, int length) {
    for (int position = length - 1; position >= 0; position--) {
```

```
        radix(a, data, size, position, length);
    }
}
```

3.5.3 Métricas

A análise das métricas obtidas indica que o algoritmo do *Radix Sort* possui uma alta eficiência, executando rapidamente na sua versão sequencial e obtendo um desempenho muito semelhante na sua versão paralela.

Radix Sort									
Algoritmo sequencial		Algoritmo paralelo (1 core)		Algoritmo paralelo (2 cores)		Algoritmo paralelo (3 cores)		Algoritmo paralelo (4 cores)	
Execução	Tempo (s)	Execução	Tempo (s)	Execução	Tempo (s)	Execução	Tempo (s)	Execução	Tempo (s)
1	0,151	1	0,204	1	0,164	1	0,256	1	0,160
2	0,182	2	0,181	2	0,184	2	0,274	2	0,162
3	0,280	3	0,162	3	0,268	3	0,184	3	0,182
4	0,188	4	0,225	4	0,384	4	0,210	4	0,169
5	0,234	5	0,295	5	0,336	5	0,205	5	0,179
6	0,236	6	0,261	6	0,712	6	0,198	6	0,182
7	0,187	7	0,223	7	0,437	7	0,213	7	0,195
8	0,230	8	0,195	8	0,346	8	0,205	8	0,196
9	0,169	9	0,220	9	0,356	9	0,216	9	0,198
10	0,271	10	0,232	10	0,400	10	0,230	10	0,194

Tabela 17: dados de tempo de execução do *Radix Sort*.

Radix Sort									
Algoritmo sequencial		Algoritmo paralelo (1 core)		Algoritmo paralelo (2 cores)		Algoritmo paralelo (3 cores)		Algoritmo paralelo (4 cores)	
Média	0,213	Média	0,220	Média	0,359	Média	0,219	Média	0,182
Mediana	0,209	Mediana	0,222	Mediana	0,351	Mediana	0,212	Mediana	0,182
Desvio Padrão	0,041416 90476	Desvio Padrão	0,0364	Desvio Padrão	0,145133 0769	Desvio Padrão	0,025882 23329	Desvio Padrão	0,013513 32676

Tabela 18: métricas do *Radix Sort*.

Os valores de Speedup medidos para as execuções com 1, 2 e 3 processadores são valores abaixo de 1, indicando que, nestas condições, o algoritmo possui um desempenho inferior ao algoritmo paralelo. Esse baixo desempenho ocorre devido a uma sobrecarga dos

recursos computacionais, causada pelo paralelismo na ordenação dos baldes e no próprio *Radix Sort*.

O Speedup obtido para 4 processadores foi de 1,171, representando um aumento de eficiência, porém este aumento é de apenas 17%. Sugerindo que a paralelização do algoritmo não trouxe ganhos substanciais no cenário testado.

No entanto, para entender melhor este resultado para 4 processadores, foram feitos outros testes com a versão sequencial e uma análise mais detalhada da parte do código que pode ser paralelizada revelou que apenas 20,**7%** do tempo de execução total corresponde à parte paralelizável.

Este valor justifica o baixo desempenho do algoritmo paralelo, uma vez que a parte paralelizável não representa uma porção significativa da execução do algoritmo.

Radix Sort				
Execução	Carregamento	Ordenação	Saída	% paralelizável
1	0,051792	0,031445	0,06734	0,2088300338
2	0,052284	0,031633	0,066741	0,2099656175
3	0,055961	0,030875	0,072519	0,1937498039
4	0,052979	0,032874	0,07387	0,2058188238
5	0,052979	0,030894	0,066765	0,2050876937
6	0,049599	0,030656	0,069545	0,2046461949
7	0,049666	0,031008	0,066072	0,2113038856
8	0,050016	0,031629	0,064528	0,2163805901
9	0,085883	0,038132	0,09441	0,174577086
10	0,052031	0,031719	0,064904	0,2133746821

Tabela 19: dados de tempo de execução separando a parte paralela do *Radix Sort*.

A partir destas medições, é possível aplicar a Lei de Amdahl para calcular os valores de Speedup teóricos conforme o aumento do número de processadores e os valores obtidos a partir dos testes:

Radix Sort		
Processadores	Amdahl	Obtido
1	1	0,9681528662
2	1,115650836	0,6472019465
3	1,160383961	0,9712460064
4	1,184123259	1,171161255
8	1,221611095	-

16	1,241259442	-

Tabela 20: dados teóricos do Speedup do *Radix Sort*.

Os cálculos indicam que a solução paralela com 4 processadores deve ter um Speedup próximo a 1,184, valor que se aproxima do Speedup real obtido na implementação do algoritmo. Os dados também indicam que o número máximo eficiente de processadores para paralelizar este código seria de 4, e adicionar mais processadores não resultaria em ganhos significativos no desempenho do algoritmo.

A partir desses números, é possível realizar o cálculo da eficiência do algoritmo, que mostra o quanto de cada processador foi utilizado nas execuções.

Radix Sort	
Processadores	Eficiência
1	0,9681528662
2	0,3236009732
3	0,3237486688
4	0,2927903137

Tabela 21: dados de eficiência do *Radix Sort*.

Nas execuções com 2 ou mais processadores, a eficiência do algoritmo é baixa, cerca de 30%. Esse desempenho pode ser explicado pela pequena proporção do código que pode ser paralelizada. Dessa forma, a parte paralelizada do código é distribuída entre vários processadores, o que reduz a carga de trabalho de cada um, mas também limita os ganhos de desempenho.

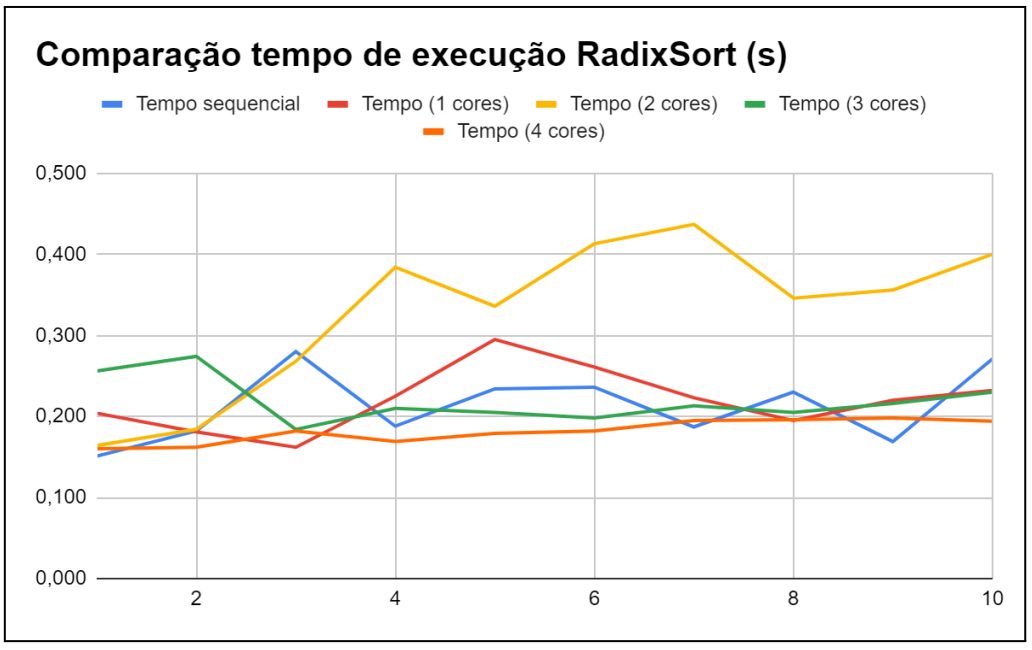


Figura 14: gráfico do desempenho do algoritmo *Radix Sort*.

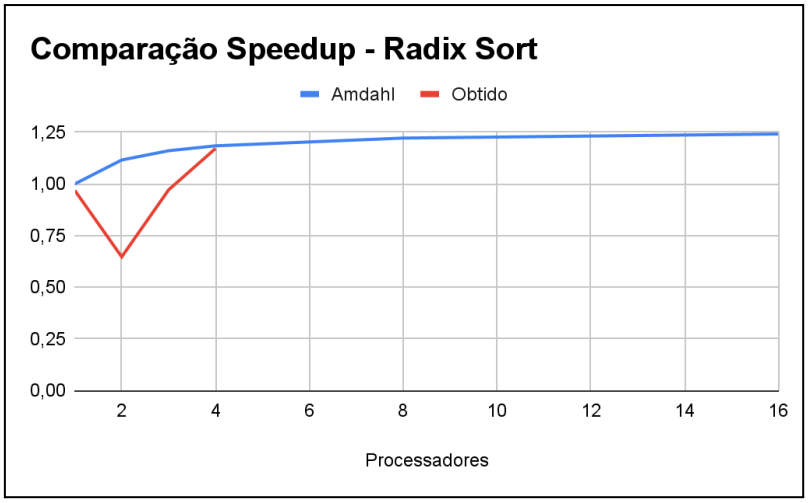


Figura 15: gráfico do cálculo de Speedup do *Radix Sort*.

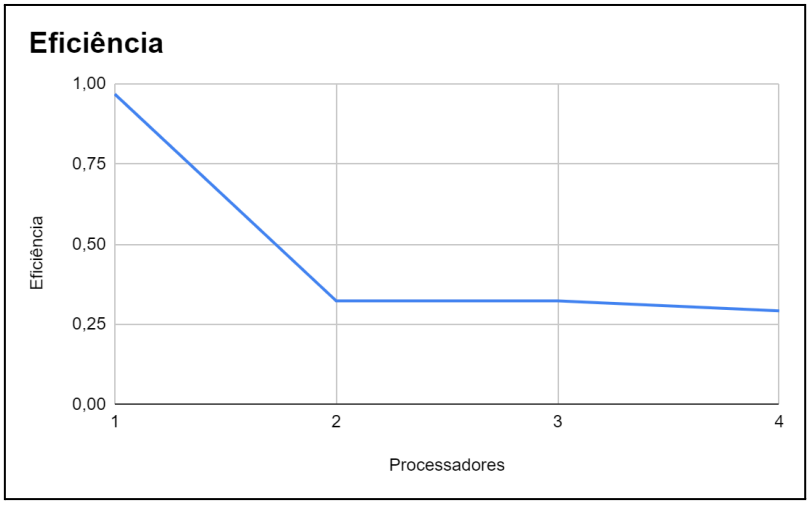


Figura 16: gráfico do cálculo de eficiência do *Radix Sort*.

4. Escalabilidade e desempenho

4.1 Speedup

Os testes realizados em cada variação dos algoritmos permitiram calcular o Speedup com base na média de 10 execuções sequenciais e 10 execuções paralelas com diferentes cores. Os resultados obtidos foram os seguintes para 4 processadores (versão mais eficiente testada):

Comparação de Speedup				
	Merge Sort	Radix Sort	Quick Sort	Odd Even Sort
Speedup	1,069861173	1,171161255	3,065459051	3,127129349

Tabela 22: comparação entre os valores de Speedup obtidos.

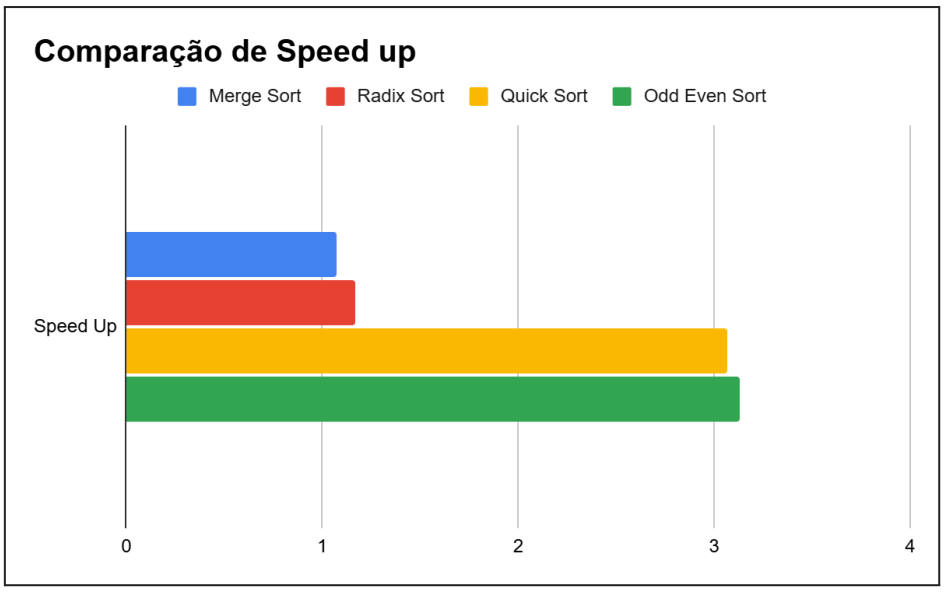


Figura 17: gráfico comparativo do Speedup.

Os algoritmos que utilizaram o *Merge Sort* e o *Radix Sort* apresentam um baixo ganho de tempo, com a diferença entre as execuções sequenciais e paralelas sendo quase nula. Entretanto, este comportamento foi atribuído ao tamanho da entrada de dados utilizada, uma vez que o tempo de execução da parte paralela dos algoritmos representava apenas 38,5% no caso do Merge Sort e 20,7% no caso do Radix Sort, em relação ao tempo total de execução. Como a maior parte do tempo de execução é dedicada à parte sequencial do algoritmo, a paralelização não gerou um ganho expressivo de desempenho.

Dessa forma, o tempo gasto pela parte sequencial acaba sendo o principal gargalo. A partir disso, com o auxílio da Lei de Amdahl, foi possível estimar o limite máximo de processadores em que o Speedup poderia aumentar de forma significativa. Esse limite se aproximou dos valores de Speedup observados nos testes.

Para validar esta teoria, seria necessário testar os princípios da Lei de Gustafson, aumentando progressivamente a entrada de dados, permitindo observar se esse aumento, proporcional ao aumento no número de processadores, resulta em um ganho de desempenho mais expressivo.

Por outro lado, os algoritmos do *Odd-Even Sort* e do *Quick Sort* apresentaram um Speedup superior que 3, evidenciando que as suas versões paralelas são expressivamente mais performáticas do que as versões sequenciais.

Todos os valores de Speedup obtidos se mostraram próximos aos valores teóricos calculados, que confirma que os algoritmos desenvolvidos possuem um alto grau de paralelismo. Porém, todos os Speedup estavam abaixo do dos valores esperados, o que sugere que existem pontos de paralelismo que ainda não foram completamente explorados, como a parte de distribuição dos dados nos baldes, que está sendo realizada de forma sequencial.

4.2 Eficiência

A eficiência dos algoritmos pode ser medida pelo número de Speedup obtido e da quantidade de processadores utilizados e é responsável por determinar o uso efetivo dos processadores alocados. Para realizar os cálculos, foram utilizados os valores de Speedup relacionados a 4 processadores, já que esta configuração se mostrou mais eficiente.

Os dados de eficiência obtidos foram:

Comparação de eficiência				
	Merge Sort	Radix Sort	Quick Sort	Odd Even Sort
Eficiência	0,2674652933	0,2927903137	0,7663647627	0,7817823373

Tabela 23: comparação entre os valores de eficiência obtidos.

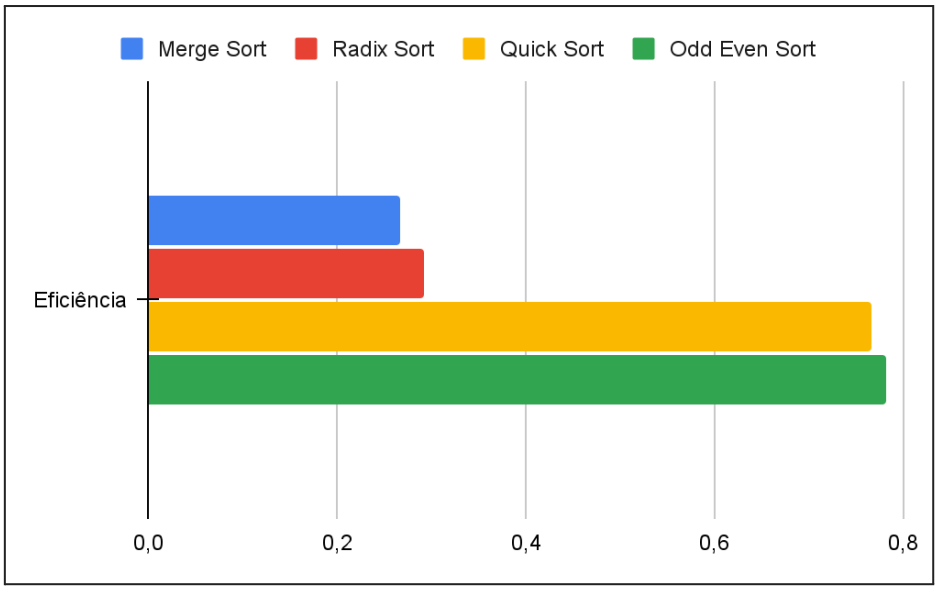


Figura 18: gráfico comparativo da eficiência.

O ideal é que os valores de eficiência estejam o mais próximo possível de 100%, indicando que todos os processadores alocados estão usando sua capacidade total.

As medições indicam que os algoritmos utilizando o Odd-Even Sort e o Quick Sort apresentaram uma eficiência semelhante, ambas se aproximaram de 80%. Isso sugere que esses algoritmos paralelos fazem um bom uso dos recursos computacionais disponíveis, aproveitando adequadamente o paralelismo e apresentando alta escalabilidade.

Por outro lado, os algoritmos com Merge Sort e Radix Sort apresentaram uma eficiência baixa, em torno de 25%, o que indica uma distribuição inadequada dos recursos computacionais. Essa baixa eficiência pode ser explicada pela característica desses

algoritmos, que possuem um tempo gasto na parte paralela proporcionalmente menor em relação ao tempo da parte sequencial, o que limita os ganhos de desempenho.

Essa teoria da má distribuição dos recursos poderia ser validada por meio de uma bateria de testes aplicando a Lei de Gustafson, que permite uma análise mais detalhada do comportamento do paralelismo com diferentes entradas que variam proporcionalmente à quantidade de processadores.

5. Solução final

Com base nas análises de desempenho realizadas dos algoritmos paralelos, é possível concluir que cada algoritmo possui características próprias que influenciam diretamente na sua paralelização e eficiência. Apesar dessas diferenças, algumas conclusões gerais podem ser feitas sobre o desempenho observado durante os testes.

Os algoritmos que utilizam o *Quick Sort* e o *Odd-Even Sort* apresentaram um Speedup significativo, com ganhos de 3 vezes em comparação com suas versões sequenciais. Esse desempenho indica que os algoritmos possuem um bom potencial para paralelização, aproveitando a adição de processadores.

Porém, eles são consideravelmente mais lentos quando comparados com o *Merge Sort* e o *Radix Sort,* que apresentaram limitações de paralelização, ambos com um ganho menor de Speedup. No entanto, a tendência destes algoritmos é apresentar uma maior escalabilidade quando testados em conjuntos maiores de dados.

Portanto, conclui-se que, mesmo com índices menores de paralelismo, o uso do *Bucket Sort* aliado ao *Merge Sort* ou ao *Radix Sort* é mais vantajoso em termos de tempo de execução e de escalabilidade. O *Quick Sort* provavelmente também teria um bom desempenho com entradas do seu caso médio, mas para comprovar seriam necessários mais testes.

5.1 Desenvolvimento do Bash

Durante o desenvolvimento do trabalho, identificou-se a necessidade de uma ferramenta que auxiliasse na medição de desempenho dos algoritmos de forma eficiente e uniforme. Neste contexto, o desenvolvimento de um script Bash trouxe praticidade e precisão ao processo, garantindo que os testes fossem conduzidos de maneira consistente entre todos os integrantes da equipe.

Esse script automatiza as execuções do código C e a medição dos tempos de execução em milissegundos, registrando os resultados em um arquivo CSV. Esse formato estruturado não só facilita a análise estatística dos dados, como também permite a fácil importação dos

resultados para ferramentas de análise de dados, oferecendo suporte para a criação de gráficos e relatórios detalhados sobre o desempenho dos algoritmos.

Em resumo, o script Bash contribuiu significativamente para a eficiência e confiabilidade do trabalho, reduzindo o risco de erro humano, acelerando o processo de coleta de dados, e possibilitando a reprodutibilidade dos experimentos. Possibilitando que o foco principal da equipe estivesse na análise dos resultados, ao invés de na execução manual dos testes.

6. Referências

DAWSON, Nathan. **Unraveling QuickSort**: the fast and versatile sorting algorithm. The Fast and Versatile Sorting Algorithm. 2023. Disponível em: https://medium.com/@nathaldawson/unraveling-quicksort-the-fast-and-versatile-sorting-algorithm-2c1214755ce9. Acesso em: 24 nov. 2024.

GEEKS FOR GEEKS. Radix Sort – Data Structures and Algorithms Tutorials. 2024. Disponível em: https://www.geeksforgeeks.org/radix-sort/. Acesso em: 24 nov. 2024.

GEEKS FOR GEEKS. Odd-Even Sort / Brick Sort. 2023. Disponível em: https://www.geeksforgeeks.org/odd-even-sort-brick-sort/. Acesso em: 24 nov. 2024.

IME USP. **Mergesort**: ordenação por intercalação. Ordenação por intercalação. 2019. Disponível em: https://www.ime.usp.br/~pf/algoritmos/aulas/mrgsrt.html. Acesso em: 24 nov. 2024.

IME USP. **Ordenação**: quicksort. Quicksort. 2020. Disponível em: https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/quick.html . Acesso em: 24 nov. 2024.

MILLS, Bryan. Parallel Sort and More Open MP: chapter 5.6. Chapter 5.6. Disponível em: https://people.cs.pitt.edu/~bmills/docs/teaching/cs1645/lecture_par_sort.pdf. Acesso em: 24 nov. 2024.

SAH, Rohit. Parallelizing Sorting Algorithms using OpenMP. 2024. Disponível em: https://dev.to/sahrohit/parallelizing-sorting-algorithms-using-openmp-1hec. Acesso em: 24 nov. 2024.[9]

STEVENARD, Jb. **Merge Sort**: merge sort is a ⁖divide and conquer⁖ algorithm; it⁖s time efficient and my favorite.. Merge Sort is a “Divide and Conquer” algorithm; it’s time efficient and my favorite.. 2022. Disponível em: https://levelup.gitconnected.com/merge-sort-656f8ee59d83. Acesso em: 24 nov. 2024

9

7. Códigos

7.1 Odd-Even final

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <omp.h>

typedef struct {
   long int *data;
   int length;
   long int total;
} bucket;

void sorted(char *a, int length, long int size) {
  for(long int i = 0; i < size - 1; i++) {
    if(strcmp(a + i * length, a + (i + 1) * length) > 0) {
      printf("não ordenou");
    }
  }
}

void odd_even(char *a, long int *data, int length, int n) {
   int sorted = 0;

   while(!sorted) {
      sorted = 1;

      #pragma omp parallel for shared(a, data, n, length, sorted) default(none)
      // Odd
      for(int i = 1; i < n - 1; i += 2) {
         if(strcmp(a + (data[i] * length), a + (data[i + 1] * length)) > 0) {
            long int temp = data[i];
            data[i] = data[i + 1];
            data[i + 1] = temp;
            sorted = 0;
         }
      }

      #pragma omp parallel for shared(a, data, n, length, sorted) default(none)
      // Even
      for(int i = 0; i < n - 1; i += 2) {
         if(strcmp(a + (data[i] * length), a + (data[i + 1] * length)) > 0) {
            long int temp = data[i];
```

```
            data[i] = data[i + 1];
            data[i + 1] = temp;
            sorted = 0;
          }
        }
      }
    }

void sort(char *a, bucket *bucket) {
    odd_even(a, bucket->data, bucket->length, bucket->total);
}

long int* bucket_sort(char *a, int length, long int size, int offset, int nbuckets) {
    long int i;
    bucket   *buckets = NULL,
           *b = NULL;
    long int *returns = NULL;


    returns = (long int *) malloc(sizeof(long int) * size);
    buckets = (bucket *) malloc(sizeof(bucket) * nbuckets);

    for (i = 0; i < nbuckets; i++) {
        buckets[i].data = returns + i * size / nbuckets;
        buckets[i].length = length;
        buckets[i].total = 0;
    }

    for (i = 0; i < size; i++) {
        b = &buckets[*(a + i * length) - offset];
        b->data[b->total++] = i;
    }

    #pragma omp parallel for shared(a, buckets)
    for (i = 0; i < nbuckets; i++)
        sort(a, &buckets[i]);

    return returns;
}

void load(char **vString, long int *N, int *l, int *os, int *nb) {
    char *strings = NULL;
    int len, offset, nbucket;
    long int n;

    if (!fscanf(stdin, "%d", &len)) {
        printf("ERROR len\n");
        exit(1);
    }
```

```
    if (!fscanf(stdin, "%ld", &n)) {
        printf("ERROR n\n");
        exit(1);
    }

    if (!fscanf(stdin, "%d", &offset)) {
        printf("ERROR offset\n");
        exit(1);
    }

    if (!fscanf(stdin, "%d", &nbucket)) {
        printf("ERROR nbucket\n");
        exit(1);
    }

    nbucket = nbucket + 1;
    len = len + 1;

    strings = (char*) malloc(n * len);

    for (int i = 0; i < n; i++) {
        if (!fscanf(stdin, "%s", strings + (i * len))) {
            printf("ERROR %d\n", i);
            exit(1);
        }
    }

    *vString = strings;
    *N = n;
    *l = len;
    *os = offset;
    *nb = nbucket;
}

void save(char *strings, long int *index, long int n, int len) {
    FILE *file = fopen("output.txt", "w");
    for (int i = 0; i < n; i++) {
        fprintf(file, "%s\n", strings + (index[i] * len));
    }
}

int main(int ac, char **av) {
    char *strings = NULL;
    long int N = 0;
    long int *index = NULL;
    int len = 0, offset, nbuckets;

    double startTime, endTime;
```

```
    load(&strings, &N, &len, &offset, &nbuckets);

    // Start timing
    startTime = omp_get_wtime();

    index = bucket_sort(strings, len, N, offset, nbuckets);

    // End timing
    endTime = omp_get_wtime();

    save(strings, index, N, len);

    //sorted(strings, len, N);

    printf("Execution time: %f seconds\n", endTime - startTime);

    if (strings != NULL)
        free(strings);

    if (index != NULL)
        free(index);

    return EXIT_SUCCESS;

}
```

7.2 Merge Sort final

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <omp.h>

typedef struct {
    long int *data;
    int length;
    long int total;
} bucket;

void sorted(char *a, int length, long int size) {
   for(long int i = 0; i < size - 1; i++) {
      if(strcmp(a + i * length, a + (i + 1) * length) > 0) {
         printf("não ordenou");
      }
   }
}

void merge(char *a, long int *data, int left, int middle, int right, int length) {
```

```
    long int n1 = middle - left + 1;
    long int n2 = right - middle;

    long int *L = (long int *) malloc(n1 * sizeof(long int));
    long int *R = (long int *) malloc(n2 * sizeof(long int));

    for (int i = 0; i < n1; i++) {
        L[i] = data[left + i];
    }

    for (int j = 0; j < n2; j++) {
        R[j] = data[middle + 1 + j];
    }

    long int i = 0, j = 0, k = left;

    while (i < n1 && j < n2) {
        if (strcmp(a + (L[i] * length), a + (R[j] * length)) < 0) {
            data[k] = L[i];
            i++;
        } else {
            data[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        data[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        data[k] = R[j];
        j++;
        k++;
    }

    free(L);
    free(R);
}

void mergesort(char *a, long int *data, int left, int right, int length, int depth) {
    if(left < right) {
        int middle = (left + right) / 2;

        if(depth < 3) {
            #pragma omp task shared(a, data, length)
```

```
            mergesort(a, data, left, middle, length, depth + 1);
            #pragma omp task shared(a, data, length)
            mergesort(a, data, middle + 1, right, length, depth + 1);
        } else {
            mergesort(a, data, left, middle, length, depth + 1);
            mergesort(a, data, middle + 1, right, length, depth + 1);
        }

        #pragma omp taskwait
        merge(a, data, left, middle, right, length);
    }
}

void sort(char *a, bucket *bucket) {
    mergesort(a, bucket->data, 0, bucket->total - 1, bucket->length, 0);
}

long int* bucket_sort(char *a, int length, long int size, int offset, int nbuckets) {
    long int i;
    bucket   *buckets = NULL,
             *b = NULL;
    long int *returns = NULL;


    returns = (long int *) malloc(sizeof(long int) * size);
    buckets = (bucket *) malloc(sizeof(bucket) * nbuckets);

    for (i = 0; i < nbuckets; i++) {
        buckets[i].data = returns + i * size / nbuckets;
        buckets[i].length = length;
        buckets[i].total = 0;
    }

    for (i = 0; i < size; i++) {
        b = &buckets[*(a + i * length) - offset];
        b->data[b->total++] = i;
    }

    #pragma omp parallel for shared(a, buckets)
    for (i = 0; i < nbuckets; i++)
        sort(a, &buckets[i]);

    return returns;
}

void load(char **vString, long int *N, int *l, int *os, int *nb) {
    char *strings = NULL;
    int len, offset, nbucket;
    long int n;
```

```
    if (!fscanf(stdin, "%d", &len)) {
        printf("ERROR len\n");
        exit(1);
    }

    if (!fscanf(stdin, "%ld", &n)) {
        printf("ERROR n\n");
        exit(1);
    }

    if (!fscanf(stdin, "%d", &offset)) {
        printf("ERROR offset\n");
        exit(1);
    }

    if (!fscanf(stdin, "%d", &nbucket)) {
        printf("ERROR nbucket\n");
        exit(1);
    }

    nbucket = nbucket + 1;
    len = len + 1;

    strings = (char*) malloc(n * len);

    for (int i = 0; i < n; i++) {
        if (!fscanf(stdin, "%s", strings + (i * len))) {
            printf("ERROR %d\n", i);
            exit(1);
        }
    }

    *vString = strings;
    *N = n;
    *l = len;
    *os = offset;
    *nb = nbucket;
}

void save(char *strings, long int *index, long int n, int len) {
    FILE *file = fopen("output.txt", "w");
    for (int i = 0; i < n; i++) {
        fprintf(file, "%s\n", strings + (index[i] * len));
    }
}

int main(int ac, char **av) {
    char *strings = NULL;
```

```
    long int N = 0;
    long int *index = NULL;
    int len = 0, offset, nbuckets;

    double startTime, endTime;

    int num_procs = omp_get_num_procs();
    printf("Number of processors: %d\n", num_procs);

    load(&strings, &N, &len, &offset, &nbuckets);

    startTime = omp_get_wtime();

    index = bucket_sort(strings, len, N, offset, nbuckets);

    endTime = omp_get_wtime();

    save(strings, index, N, len);

    //sorted(strings, len, N);

    printf("Execution time: %f seconds\n", endTime - startTime);

    free(strings);
    free(index);

    return EXIT_SUCCESS;
}
```

7.3 Quick Sort final

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct {
  long int *data;
  int length;
  long int total;
} bucket;

void sorted(char *a, int length, long int size) {
  for (long int i = 0; i < size - 1; i++) {
    if (strcmp(a + i * length, a + (i + 1) * length) > 0) {
      printf("não ordenou");
    }
  }
```

```
}

int partition(char *a, long int *data, int low, int high, int length) {
 long int pivot = data[high];
 int i = low - 1;

 for (int j = low; j < high; j++) {
  if (strcmp(a + data[j] * length, a + pivot * length) <= 0) {
   i++;
   long int temp = data[i];
   data[i] = data[j];
   data[j] = temp;
  }
 }

 long int temp = data[i + 1];
 data[i + 1] = data[high];
 data[high] = temp;

 return i + 1;
}

void quicksort(char *a, long int *data, int low, int high, int length, int depth) {
 if (low < high) {
  int pi = partition(a, data, low, high, length);

  if (depth < 3) {
#pragma omp task shared(a, data, length) if (high - low > 1000)
   quicksort(a, data, low, pi - 1, length, depth + 1);

#pragma omp task shared(a, data, length) if (high - low > 1000)
   quicksort(a, data, pi + 1, high, length, depth + 1);

#pragma omp taskwait
  } else {
   quicksort(a, data, low, pi - 1, length, depth + 1);
   quicksort(a, data, pi + 1, high, length, depth + 1);
  }
 }
}

void sort(char *a, bucket *bucket) {
 #pragma omp parallel
 {
  #pragma omp single
  quicksort(a, bucket->data, 0, bucket->total - 1, bucket->length, 0);
 }
}
```

```
long int *bucket_sort(char *a, int length, long int size, int offset,
                      int nbuckets) {
  long int i;
  bucket *buckets = NULL;
  long int *returns = NULL;

  returns = (long int *)malloc(sizeof(long int) * size);
  buckets = (bucket *)malloc(sizeof(bucket) * nbuckets);

  for (i = 0; i < nbuckets; i++) {
    buckets[i].data = returns + i * size / nbuckets;
    buckets[i].length = length;
    buckets[i].total = 0;
  }

  for (i = 0; i < size; i++) {
    bucket *b = &buckets[*(a + i * length) - offset];
    b->data[b->total++] = i;
  }


  #pragma omp parallel for shared(a, buckets)
  for (i = 0; i < nbuckets; i++) {
    sort(a, &buckets[i]);
  }

  free(buckets);
  return returns;
}

void load(char **vString, long int *N, int *l, int *os, int *nb) {
  char *strings = NULL;
  int len, offset, nbucket;
  long int n;

  if (!fscanf(stdin, "%d", &len)) {
    printf("ERROR len\n");
    exit(1);
  }

  if (!fscanf(stdin, "%ld", &n)) {
    printf("ERROR n\n");
    exit(1);
  }

  if (!fscanf(stdin, "%d", &offset)) {
    printf("ERROR offset\n");
    exit(1);
  }
```

```
  if (!fscanf(stdin, "%d", &nbucket)) {
    printf("ERROR nbucket\n");
    exit(1);
  }

  nbucket = nbucket + 1;
  len = len + 1;

  strings = (char *)malloc(n * len);

  for (int i = 0; i < n; i++) {
    if (!fscanf(stdin, "%s", strings + (i * len))) {
      printf("ERROR %d\n", i);
      exit(1);
    }
  }

  *vString = strings;
  *N = n;
  *l = len;
  *os = offset;
  *nb = nbucket;
}

void save(char *strings, long int *index, long int n, int len) {
  for (int i = 0; i < n; i++) {
    printf("%s\n", strings + (index[i] * len));
  }
}

int main(int ac, char **av) {
  char *strings = NULL;
  long int N = 0;
  long int *index = NULL;
  int len = 0, offset, nbuckets;

  double startTime, endTime;

  load(&strings, &N, &len, &offset, &nbuckets);

  // Start timing
  startTime = omp_get_wtime();

  index = bucket_sort(strings, len, N, offset, nbuckets);

  // End timing
  endTime = omp_get_wtime();
```

```
 save(strings, index, N, len);

 // sorted(strings, len, N);

 printf("Execution time: %f seconds\n", endTime - startTime);

 if (strings != NULL)
   free(strings);

 if (index != NULL)
   free(index);

 return EXIT_SUCCESS;
}
```

7.4 Radix Sort final

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <omp.h>

#define MAX_RADIX 256

typedef struct {
   long int *data;
   int length;
   long int total;
} bucket;

void sorted(char *a, int length, long int size) {
  for(long int i = 0; i < size - 1; i++) {
    if(strcmp(a + i * length, a + (i + 1) * length) > 0) {
      printf("não ordenou");
    }
  }
}

void radix(char *a, long int *data, int size, int position, int length) {
   long int *sortedData = (long int *) malloc(size * sizeof(long int));
   int *RADIXCOUNT = (int *) malloc(MAX_RADIX * sizeof(int));

   for (int i = 0; i < MAX_RADIX; i++) {
      RADIXCOUNT[i] = 0;
   }

   #pragma omp parallel for
   for (int i = 0; i < size; i++) {
```

```
        int valorPos = a[data[i] * length + position];
        RADIXCOUNT[valorPos]++;
    }

    for (int i = 1; i < MAX_RADIX; i++) {
        RADIXCOUNT[i] += RADIXCOUNT[i - 1];
    }

    #pragma omp parallel for
    for (int i = size - 1; i >= 0; i--) {
        int valorPos = a[data[i] * length + position];
        sortedData[RADIXCOUNT[valorPos] - 1] = data[i];
        RADIXCOUNT[valorPos]--;
    }

    #pragma omp parallel for
    for (int i = 0; i < size; i++) {
        data[i] = sortedData[i];
    }

    free(sortedData);
    free(RADIXCOUNT);
}

void radixsort(char *a, long int *data, int size, int length) {
    for (int position = length - 1; position >= 0; position--) {
        radix(a, data, size, position, length);
    }
}

void sort(char *a, bucket *bucket) {
    radixsort(a, bucket->data, bucket->total, bucket->length);
}

long int* bucket_sort(char *a, int length, long int size, int offset, int nbuckets) {
    long int i;
    bucket *buckets = NULL, *b = NULL;
    long int *returns = NULL;

    returns = (long int *) malloc(sizeof(long int) * size);
    buckets = (bucket *) malloc(sizeof(bucket) * nbuckets);

    for (i = 0; i < nbuckets; i++) {
        buckets[i].data = returns + i * (size / nbuckets);
        buckets[i].length = length;
        buckets[i].total = 0;
    }

    for (i = 0; i < size; i++) {
```

```
        b = &buckets[*(a + i * length) - offset];
        b->data[b->total++] = i;
    }

    // sort each "bucket"
    //#pragma omp task shared(a, buckets)
    #pragma omp parallel for shared(a, buckets)
    for (i = 0; i < nbuckets; i++)
        sort(a, &buckets[i]);

    return returns;
}

void load(char **vString, long int *N, int *l, int *os, int *nb) {
    char *strings = NULL;
    int len, offset, nbucket;
    long int n;

    if (!fscanf(stdin, "%d", &len)) {
        printf("ERROR len\n");
        exit(1);
    }

    if (!fscanf(stdin, "%ld", &n)) {
        printf("ERROR n\n");
        exit(1);
    }

    if (!fscanf(stdin, "%d", &offset)) {
        printf("ERROR offset\n");
        exit(1);
    }

    if (!fscanf(stdin, "%d", &nbucket)) {
        printf("ERROR nbucket\n");
        exit(1);
    }

    nbucket = nbucket + 1;
    len = len + 1;

    strings = (char*) malloc(n * len);

    for (int i = 0; i < n; i++) {
        if (!fscanf(stdin, "%s", strings + (i * len))) {
            printf("ERROR %d\n", i);
            exit(1);
        }
    }
```

```
    *vString = strings;
    *N = n;
    *l = len;
    *os = offset;
    *nb = nbucket;
}

void save(char *strings, long int *index, long int n, int len) {
    FILE *file = fopen("output.txt", "w");
    for (int i = 0; i < n; i++) {
        fprintf(file, "%s\n", strings + (index[i] * len));
    }
}

int main(int ac, char **av) {

    char *strings = NULL;
    long int N = 0;
    long int *index = NULL;
    int len = 0, offset, nbuckets;

    double startTime, endTime;

    load(&strings, &N, &len, &offset, &nbuckets);

    // Start timing
    startTime = omp_get_wtime();

    index = bucket_sort(strings, len, N, offset, nbuckets);

    // End timing
    endTime = omp_get_wtime();

    save(strings, index, N, len);

   // sorted(strings, len, N);

    printf("Execution time: %f seconds\n", endTime - startTime);

    if (strings != NULL)
        free(strings);

    if (index != NULL)
        free(index);

    return EXIT_SUCCESS;
}
```

7.5 Bash

```
#!/bin/bash


# Para rodar esse arquivo, use o comando:
# ./teste.sh <caminho_do_codigo.c> <numero_de_execucoes> <numero_de_cores>
# EX: ./teste.sh mergesort/mergesort.c 5 4
# Isso roda mergesort.c que está na pasta mergesort 5 vezes, usando 4 cores.


# Caminho para o arquivo C a ser compilado
codigo_c=$1
# Número de execuções
num_execucoes=$2
# Número de cores
num_cores=$3
# Arquivo de entrada
input_file="input"


# Verifica se os parâmetros foram passados
if [[ -z $codigo_c || -z $num_execucoes || -z $num_cores ]]; then
        echo "Uso correto: ./teste.sh <caminho_do_codigo.c> <numero_de_execucoes> <numero_de_cores>"
    exit 1
fi


# Verifica se o arquivo C existe
if [[ ! -f $codigo_c ]]; then
```

```
    echo "Arquivo de código '$codigo_c' não encontrado!"
    exit 1
fi

# Verifica se o arquivo de entrada 'input' existe
if [[ ! -f $input_file ]]; then
    echo "Arquivo de entrada '$input_file' não encontrado!"
    exit 1
fi

# Extrai o diretório do arquivo C e o nome base do arquivo
diretorio_codigo=$(dirname "$codigo_c")
nome_codigo=$(basename "$codigo_c" .c)

# Cria o caminho do arquivo de log em formato CSV, incluindo o número de cores no nome
log_file="$diretorio_codigo/${nome_codigo}_cores_${num_cores}_log.csv"

# Cria o arquivo CSV com cabeçalhos
echo "Execução,Tempo (s)" > "$log_file"

# Compila o código C especificado, usando o nome do arquivo C como nome do executável
echo "Compilando '$codigo_c' com $num_cores cores..."
gcc "$codigo_c" -o "$diretorio_codigo/$nome_codigo" -O2 -fopenmp -lm

# Verifica se a compilação foi bem-sucedida
```

```
    if [[ $? -ne 0 ]]; then
        echo "Erro na compilação do código C."
        exit 1
    fi

    # Configura o número de threads OpenMP
    export OMP_NUM_THREADS=$num_cores

    # Loop para executar o programa N vezes e calcular a soma dos tempos
    echo "Executando o teste $num_execucoes vezes com $num_cores cores..."
    soma_tempos=0
    for ((i = 1; i <= num_execucoes; i++)); do
        echo "Executando o teste número $i..."
        # Usa o comando time para registrar o tempo
        tempo_execucao=$( { time "$diretorio_codigo/$nome_codigo" < "$input_file"; } 2>&1
| grep real | awk '{print $2}' )

        # Separa minutos e segundos usando regex
        if [[ $tempo_execucao =~ ([0-9]+)m([0-9]+)\.([0-9]+)s ]]; then
            minutos=${BASH_REMATCH[1]}
            segundos=${BASH_REMATCH[2]}
            milissegundos=${BASH_REMATCH[3]}
            # Converte para segundos com precisão decimal
            tempo_total=$(awk -v m="$minutos" -v s="$segundos" -v ms="$milissegundos"
'BEGIN {printf "%.3f", m * 60 + s + ms / 1000}')
        else
```

```
        tempo_total=$(echo "$tempo_execucao" | sed 's/,/./') # Usa o formato padrão de ponto decimal
    fi

    # Soma os tempos
    soma_tempos=$(awk -v soma="$soma_tempos" -v atual="$tempo_total" 'BEGIN {printf "%.3f", soma + atual}')

    # Registra no arquivo CSV
    echo "$i,$tempo_total" >> "$log_file"
    echo "Tempo da execução $i: ${tempo_total}s"
done

# Calcula a média usando awk
media=$(awk -v soma="$soma_tempos" -v execs="$num_execucoes" 'BEGIN {printf "%.3f", soma / execs}')

# Exibe a média no terminal
echo "--------------------------------"
echo "Média do tempo de execução: ${media}s"
echo "Número de cores utilizados: $num_cores"

# Adiciona as informações de média e número de cores ao final do arquivo CSV
echo "Média,$media" >> "$log_file"
echo "Número de cores,$num_cores" >> "$log_file"
```