



UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
INSTITUTO METRÓPOLE DIGITAL
TÓPICOS ESPECIAIS EM INTERNET DAS COISAS "B" - **IMD291**

ANÁLISE DE CÓDIGOS PARALELOS E SERIAIS

ODD-EVEN TRANSPOSITION SORT

Prof. Dr. Kayo Gonçalves e Silva
Discente: Oziel Alves do Nascimento Júnior
Matrícula: 20170065711

Natal, 2020

Análise de Algoritmos Paralelos e Seriais

Odd-Even Transposition Sort

Universidade Federal do Rio Grande do Norte ([UFRN](#)), 2020.

Análise por:

[Oziel Alves](#)

Esta análise se encontra disponível em:

<https://github.com/ozielalves/prog-paralela/tree/master/OETSort>

Sumário

- [Introdução](#)
 - [Objetivos](#)
 - [Dependências](#)
 - [G++ Compiler](#)
 - [MPI](#)
 - [Compilação e Execução](#)
 - [Arquivo com Resultados](#)
 - [Condições de Testes](#)
 - [Informações sobre a máquina utilizada](#)
 - [Apresentação do Algoritmo](#)
 - [Odd-Even Transposition Sort](#)
 - [Serial](#)
 - [Paralelo](#)
- [Desenvolvimento](#)
 - [Corretude](#)
 - [Gráficos](#)
 - [Análise de Speedup](#)
 - [Análise de Eficiência](#)
- [Conclusão](#)
 - [Considerações Finais](#)
 - [Softwares utilizados](#)

Introdução

Objetivos

Analisar e avaliar o comportamento, eficiência e speedup dos códigos seriais e paralelos referentes a implementação do algoritmo de ordenação **Odd-Even Transposition** em relação aos tempos de execução, tamanhos de problema analisados e resultados obtidos. Os cenários irão simular a execução dos algoritmos para 1 (serial), 2, 4 e 8 cores, com 4 tamanhos de problema definidos empiricamente. Os limites de tamanhos foram estabelecidos com o objetivo de atingir o tempo mínimo de execução determinado pela [referência](#) desta Análise.

Dependências

G++ Compiler

É necessário para a compilação dos programam, visto que são feitos em c++.

```
# Instalação no Ubuntu 20.04 LTS:
sudo apt-get install g++
```

MPI - Message Passing Interface

É necessário para a compilação e execução dos códigos paralelos.

```
# Instalação no Ubuntu 20.04 LTS:
sudo apt-get install -y mpi
```

Compilação e Execução

Instaladas as dependências, basta executar o shellscrip determinado para a devida bateria de execuções na raiz do repositório: Serão realizadas **5 execuções** com **4 tamanhos de problema** , em **3 quantidades de cores** (2, 4 e 8).

```
# Para o algoritmo de ordenação serial
./OETS_serial_start.sh
```

```
# Para o algoritmo de ordenação paralelo
./OETS_paralelo_start.sh
```

Obs.: Caso seja necessário conceder permissão máxima para os scripts, execute `chmod 777 [NOME DO SCRIPT].sh` .

Arquivo com Resultados

Após o termino das execuções do script é possível ter acesso aos arquivos de tempo `.txt` na pasta `serial` ou `paralelo` , de acordo com o script executado. Os dados coletados foram utilizados para realização desta análise.

Condições de Testes

Informações sobre a máquina utilizada

- Dell Inspiron 14-inc 7460
- **Processador:** Intel Core i7 7500U (até 3.5 GHz) Dual Core Cache 4M. (FSB)4 GT/s OPI (Integra Hyper-Threading para trabalhar com até 4 threads de uma vez)
- **Número de Cores/Threads:** 2/4
- **Memória:** 8 GB tipo DDR4 – 2133MHz
- **Sistema:** Ubuntu 20.04.1 LTS

Apresentação do Algoritmo

Odd-Even Transposition Sort

O algoritmo Odd-Even Transposition ordena n elementos em n fases (n é par), cada fase requer $n / 2$ operações de troca por comparação. Esse algoritmo alterna entre duas fases, **Odd** (ímpares) e **Even** (pares). Seja $[a_1, a_2, \dots, a_n]$ uma lista a ser ordenada. Durante a fase Odd, os elementos com índices ímpares são comparados com seus vizinhos direitos e, se estiverem fora da sequência, são trocados; assim, os pares $(a_1, a_2), (a_3, a_4), \dots, (a_{n-1}, a_n)$ são trocados por comparação (assumindo que n é par). Da mesma forma, durante a fase Even, os elementos com índices pares são comparados com seus vizinhos direitos, e se eles estiverem fora de sequência, eles são trocados; assim, os pares $(a_2, a_3), (a_4, a_5), \dots, (a_{n-2}, a_{n-1})$ são trocados por comparação. Após n fases de trocas Odd-Even, a lista é ordenada. Cada fase do algoritmo (Odd ou Even) requer comparações $Q(n)$, e há um total de n fases; assim, a complexidade sequencial é $Q(n^2)$.

Referência: Introduction to Parallel Computing-Ananth Gramma (2nd Edition)

Serial

Dado um número `n` de elementos para criação de uma lista com inteiros randômicos, a seguinte sub-rotina é implementada:

1. `list` recebe `n` espaços de memória alocados referentes a lista a ser ordenada.
2. Em seguida, a função `genList` se responsabiliza por popular `list` com números inteiros pseudoaleatórios.
3. Feito isto, a função `oddEvenSort` pode dar início ao processo de ordenação de maneira análoga à descrição anterior.
4. Após o termino da ordenação, a memória alocada para `list` é liberada.

A implementação da função `oddEvenSort` é apresentada abaixo:

```
void oddEvenSort(int *list, int n)
{
    bool isSorted = false; # Flag que indica se a lista está ordenada

    while (!isSorted)
    {
        isSorted = true;

        # Fase ímpar (Odd)
        for (size_t i = 1; i <= n-2; i = i+2)
        {
            if (list[i] > list[i+1])
            {
                swap(list[i], list[i+1]);
                isSorted = false;
            }
        }

        # Fase par (Even)
        for (size_t i = 0; i <= n-2; i=i+2)
        {
            if (list[i] > list[i+1])
            {
                swap(list[i], list[i+1]);
                isSorted = false;
            }
        }
    }

    return;
}
```

Paralelo

Ainda sendo `n` o número de elementos para criação de uma lista com inteiros randômicos, a seguinte sub-rotina é implementada:

1. É iniciada a comunicação paralela.
2. Em um processo similar ao que acontece no código serial, `list` recebe `n` espaços de memória alocados referentes a lista a ser ordenada.
3. Em seguida, a função `genList` se responsabiliza por popular `list` com números inteiros pseudoaleatórios.
4. Feito isto, é chamada a função `MPI_OETS` para distribuir as parcelas de lista igualmente entre os processos, utilizando `MPI_Scatter`, e ordenar as parcelas de lista em cada processo, usando a função `oddEvenSort` já implementada no código serial.
5. Logo em seguida, é dado início à etapa de comunicação e troca de dados entre os processos utilizando a dinâmica de fases **Odd-Even**. Para auxiliar este processo é então chamada a função `MPI_SWAP` para cada 2 processos que precisam realizar a comunicação, também passando como parâmetro a lista local referente ao processo que está enviando o dado.
6. Dentro da `MPI_SWAP`, as listas locais referentes ao processo remetente e ao processo destinatário são unidas usando a função `Join`. A lista `merged_list`, resultante da união, é então ordenada, utilizando novamente a função `oddEvenSort`, para que a menor e a maior parcela sejam redistribuídas aos processos em comunicação de acordo com a hierarquia entre eles.
7. Finalizado o processo de comunicação e Troca de dados entre os processos, as listas locais são reunidas na lista principal de forma ordenada usando `MPI_Gather`.

8. Após o termino da ordenação, a memória alocada para `list` é liberada.

9. A comunicação MPI é finalizada.

A implementação do Paralelismo é apresentada abaixo:

```
/*
 * Join nas duas listas locais (Ordem não preservada)
 *
 * arg list_rcv: local_list_rcv
 * arg len_list_rcv: tamanho de local_list_rcv
 * arg list_snd: local_list_snd
 * arg len_list_snd: tamanho de local_list_snd
 * arg merged_list: lista auxiliar resultante
 */

int Join(int *local_list_rcv, int len_local_list_rcv, int *local_list_snd,
        int len_local_list_snd, int *merged_list)
{
    int i, j = 0;
    int aux = 0;

    while (i < len_local_list_rcv)
    {
        merged_list[aux++] = local_list_snd[i++];
    }
    # Sem garantia de ordem
    while (j < len_local_list_snd)
    {
        merged_list[aux++] = local_list_snd[j++];
    }

    return 0;
}

/*
 * O rank remetente envia os dados para fazer o swap e aguarda o retorno.
 * O rank destinatario recebe os dados, ordena o novo array e retorna
 * a outra metade que cabe ao rank remetente.
 */
void MPI_SWAP(int local_n, int *local_list, int snd_rank, int rcv_rank, MPI_Comm comm)
{
    int my_rank;           # Rank dos meus processos
    const int merge_id = 1; # Identifica a comunicação
    const int sorted_id = 2; # Identifica a comunicação
    int aux_list[local_n];  # Lista auxiliar cópia da lista local do rank remetente
    int merged_list[2 * local_n]; # Lista auxiliar oriunda do Merge(local_n_rcv, local_n_snd)

    MPI_Comm_rank(comm, &my_rank);

    # Na fase Odd da comunicação
    if (my_rank == snd_rank)
    {
        # A rotina é bloqueada até que o rank destinatario receba a o dado
        MPI_Send(local_list, local_n, MPI_INT, rcv_rank, merge_id, MPI_COMM_WORLD);
        # MPI Status nao necessario para esta rotina
        MPI_Recv(local_list, local_n, MPI_INT, rcv_rank, sorted_id, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
    else
    {
        # Recebe local_list referente ao rank remetente
        MPI_Recv(aux_list, local_n, MPI_INT, snd_rank, merge_id, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        # Uniao da local_list do rank destinatario com a local_list do rank remetente (aux_list)
        Join(local_list, local_n, aux_list, local_n, merged_list);

        # Ordenação pós join
        oddEvenSort(merged_list, 2 * local_n);

        int start = 0;
        int end = local_n;

        # Garante que o Maior elemento suba && Menor elemento desca
    }
}
```

```

    if (my_rank < snd_rank)
    {
        start = local_n;
        end = 0;
    }
    else
    {
        # Nada
    }

    # Envia a parte que cabe ao rementeante já ordenada, após o Merge
    MPI_Send(&(merged_list[start]), local_n, MPI_INT, snd_rank, sorted_id, MPI_COMM_WORLD);

    # Atualiza local_list
    for (int i = end; i < end + local_n; i++)
    {
        # O rank destinatario fica com os Maiores elementos
        local_list[i - end] = merged_list[i];
    }
}

}

# Ordenação local seguida de comunicacao-swap
void MPI_OETS(int n, int *list, MPI_Comm comm)
{
    int my_rank, p, i;
    int root_rank = 0;
    int *local_list; # Lista local

    MPI_Comm_rank(comm, &my_rank);
    MPI_Comm_size(comm, &p);

    local_list = (int *)calloc(n / p, sizeof(int)); # Aloca e inicializa

    # Divide list em partes iguais "local_list" para cada processo
    MPI_Scatter(list, n / p, MPI_INT, local_list, n / p, MPI_INT, root_rank, comm);

    # Ordena a lista local
    oddEvenSort(local_list, n / p);

    # Permutação Odd - Even
    for (i = 1; i <= p; i++)
    {
        # Fase ímpar (Odd)
        if ((my_rank + i) % 2 == 0)
        {
            if (my_rank < p - 1)
            {
                MPI_SWAP(n / p, local_list, my_rank, my_rank + 1, comm);
            }
        }
        # Fase par (Even)
        else if (my_rank > 0)
        {
            MPI_SWAP(n / p, local_list, my_rank - 1, my_rank, comm);
        }
        else
        {
            # Nada
        }
    }

    # Reune cada local_list na lista principal agora de forma ordeanda
    MPI_Gather(local_list, n / p, MPI_INT, list, n / p, MPI_INT, root_rank, comm);
}

```

Obs.: Em uma tentativa de otimizar o algoritmo foi feita uma mudança na etapa 5 da sub-rotina implementada para a ordenação da lista. A função `Join`, utilizada nesta etapa, foi substituída pela função `Merge` que realiza a união das duas listas locais preservando a ordenação já existente nestas parcelas, o que descarta a necessidade de utilizar novamente a função `oddEvenSort`. Esta técnica tem como propósito a redução do tempo gasto em mais uma ordenação para cada comunicação realizada. A eficiência obtida a partir desta decisão será melhor discutida no item [Desenvolvimento](#). Observe abaixo a implementação da função `Merge`:

```

bash
/*
 * Merge nas duas listas locais de forma a preservar a ordenação
 *
 * arg list_rcv: local_list_rcv
 * arg len_list_rcv: tamanho de local_list_rcv
 * arg list_snd: local_list_snd
 * arg len_list_snd: tamanho de local_list_snd
 * arg merged_list: lista auxiliar resultante
 */

int Merge(int *local_list_rcv, int len_local_list_rcv, int *local_list_snd,
          int len_local_list_snd, int *merged_list)
{
    int i, j;
    int aux = 0;

    for (i = 0, j = 0; i < len_local_list_rcv; i++)
    {
        # Garantia de ordem
        while ((local_list_snd[j] < local_list_rcv[i]) && j < len_local_list_snd)
        {
            merged_list[aux++] = local_list_snd[j++];
        }
        merged_list[aux++] = local_list_rcv[i];
    }
    while (j < len_local_list_snd)
    {
        merged_list[aux++] = local_list_snd[j++];
    }

    return 0;
}

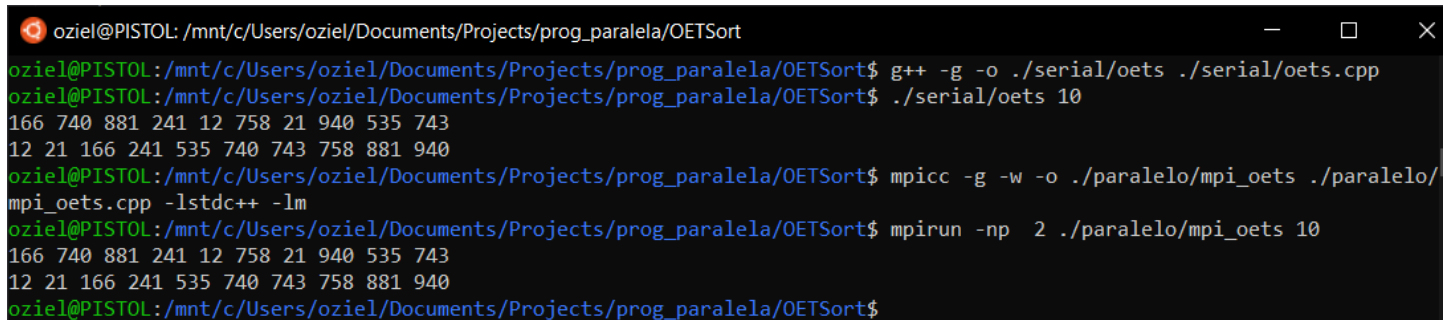
```

Desenvolvimento

Para esta análise, serão realizadas 5 execuções com tamanhos de problema 92.000, 108.000, 124.000 e 140.000 - definidos empiricamente de modo a atingir os limites mínimos determinados pela [referência](#) da análise - em 3 quantidades de cores (2, 4 e 8). Se espera que a eficiência do algoritmo paralelo quanto à ordenação da lista seja maior para um mesmo tamanho de problema quando se altera apenas o número de cores. Uma descrição completa da máquina de testes pode ser encontrada no tópico [Condições de Testes](#).

Corretude

Para validar a corretude dos algoritmos implementados foi realizado um teste utilizando 10 como tamanho de problema para os dois códigos:



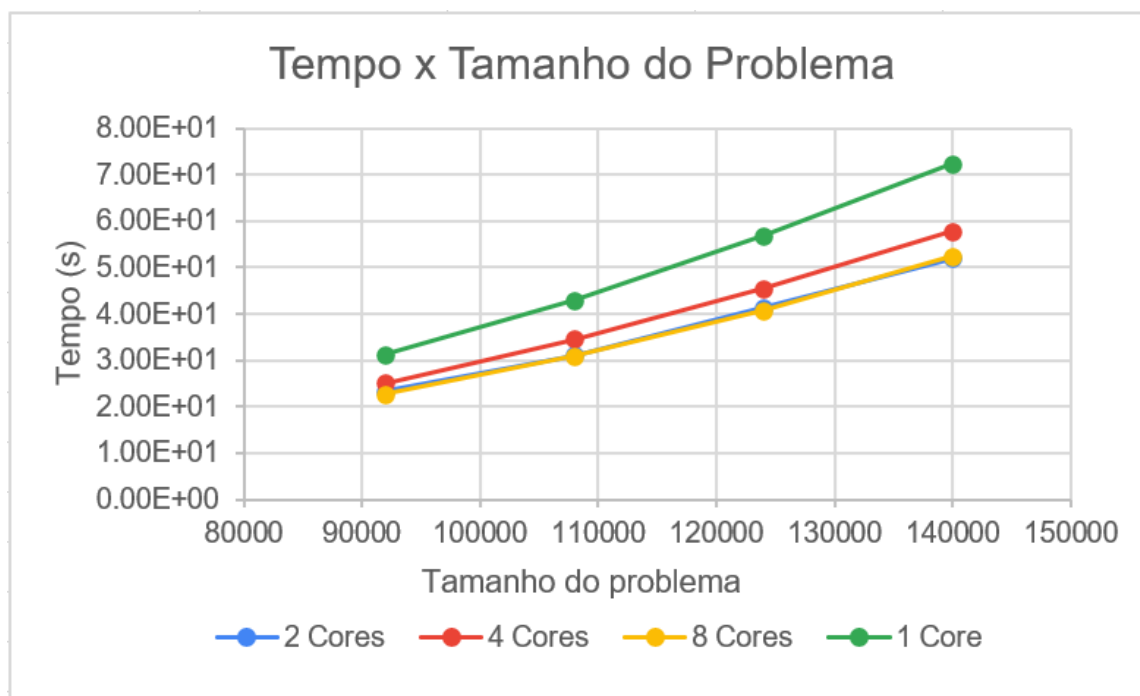
```

oziel@PISTOL: /mnt/c/Users/oziel/Documents/Projects/prog_paralela/OETSort
oziel@PISTOL:/mnt/c/Users/oziel/Documents/Projects/prog_paralela/OETSort$ g++ -g -o ./serial/oets ./serial/oets.cpp
oziel@PISTOL:/mnt/c/Users/oziel/Documents/Projects/prog_paralela/OETSort$ ./serial/oets 10
166 740 881 241 12 758 21 940 535 743
12 21 166 241 535 740 743 758 881 940
oziel@PISTOL:/mnt/c/Users/oziel/Documents/Projects/prog_paralela/OETSort$ mpicc -g -w -o ./paralelo/mpi_oets ./paralelo/mpi_oets.cpp -lstdc++ -lm
oziel@PISTOL:/mnt/c/Users/oziel/Documents/Projects/prog_paralela/OETSort$ mpirun -np 2 ./paralelo/mpi_oets 10
166 740 881 241 12 758 21 940 535 743
12 21 166 241 535 740 743 758 881 940
oziel@PISTOL:/mnt/c/Users/oziel/Documents/Projects/prog_paralela/OETSort$

```

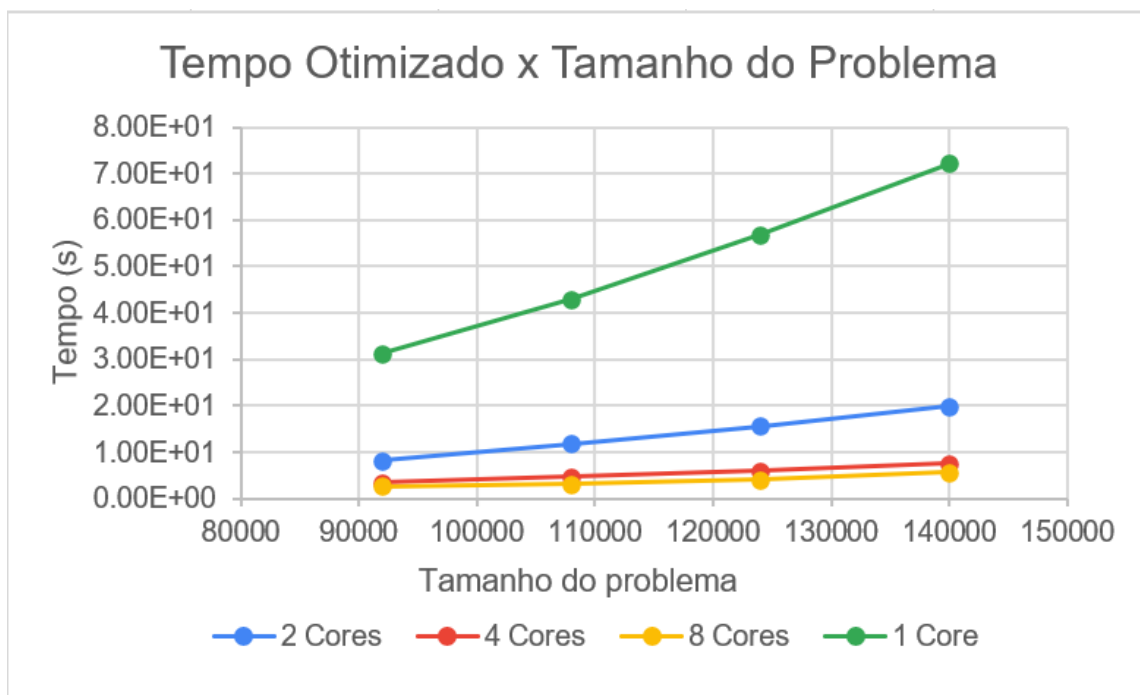
Como é possível perceber, ambos os códigos conseguem ordenar corretamente as listas geradas com *seed* única.

Gráficos



Através do gráfico de tempos por problema, é possível observar que o código paralelo consegue reduzir o tempo de execução do algoritmo se comparado ao código serial. Também é evidente o impacto da utilização da ordenação na função `MPI_SWAP`, percebe-se que o tempo de execução dos problemas não sofre redução alguma se aumentado o número de cores utilizados.

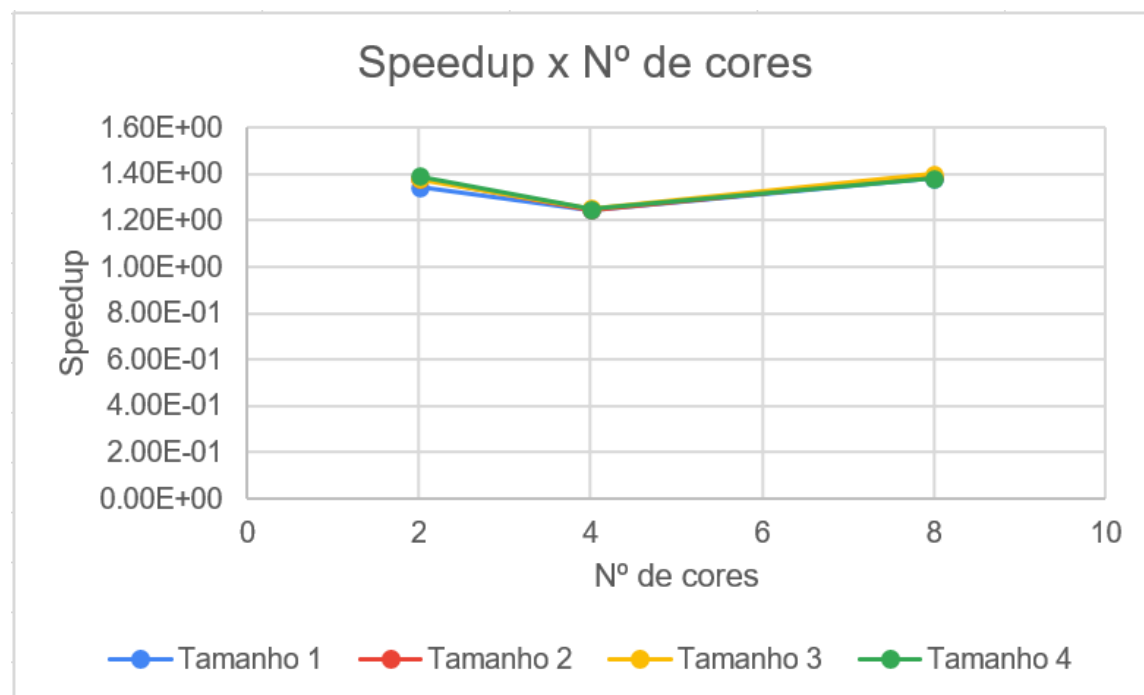
Observe agora um segundo gráfico de tempos considerando a execução do código paralelo utilizando a otimização implementada pela função `Merge` na comunicação entre os processos.



O tempo de execução para o código paralelo cai de maneira bastante considerável. Nota-se ainda que existe um aumento na velocidade da execução para todos os tamanhos de problema, de maneira proporcional ao aumento do número de cores utilizados, como era esperado em um modelo ótimo. Isso acontece, em resumo, porque a conservação da ordenação garantida pela função `Merge` economiza eventuais comparações desnecessárias.

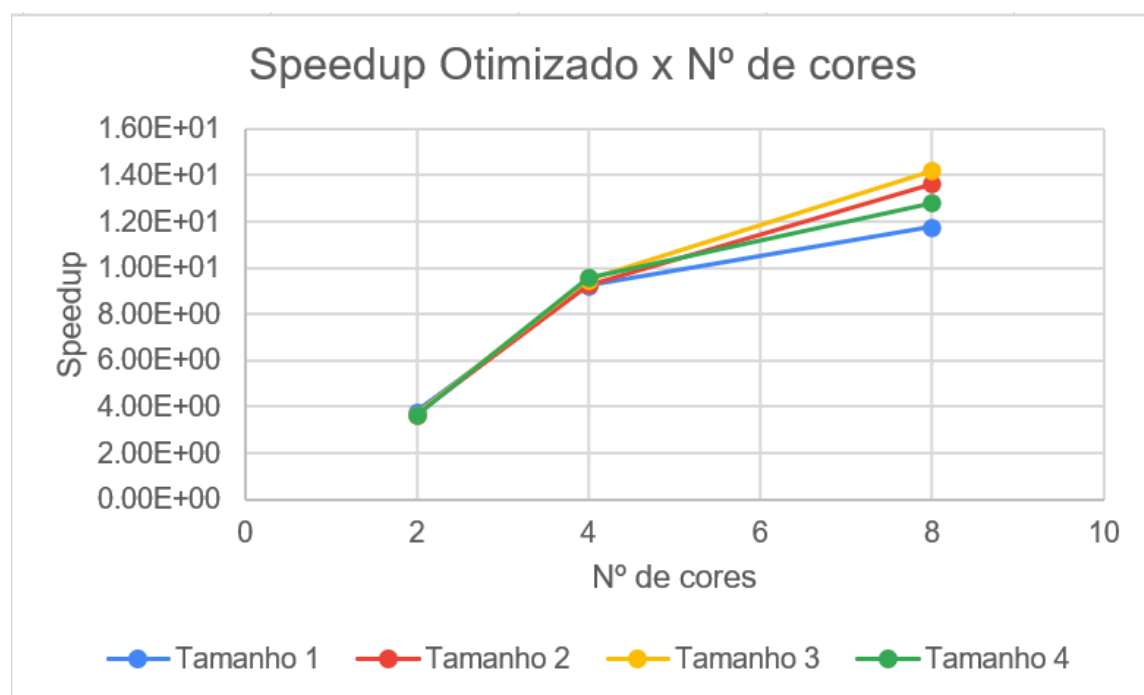
Análise de Speedup

É possível definir o *speedup*, quando da utilização de n cores, como sendo o tempo médio de execução no código serial dividido pelo tempo médio de execução para n cores em um dado tamanho de problema. Dessa forma, o speedup representa um aumento médio de velocidade na resolução dos problemas. No gráfico abaixo é possível perceber de maneira mais clara o que acontece com o speedup do código paralelo quando é preciso ordenar as parcelas de listas passadas a cada comunicação de processos.



Como foi mencionado anteriormente, fica claro que a execução dos problemas em 4 cores levou um tempo maior quando comparada a utilização de 2 cores, e que o desempenho para 8 cores se mostrou similar ao desempenho para 2 cores, isto ocorre porque o código paralelo inicial realiza também a ordenação sempre que existe uma comunicação entre os processos. Como resultado, temos um speedup médio muito similar para 2 e 8 cores e um menor speedup médio para 4 cores.

Observe agora o speedup relativo ao código paralelo otimizado pela função `Merge` :



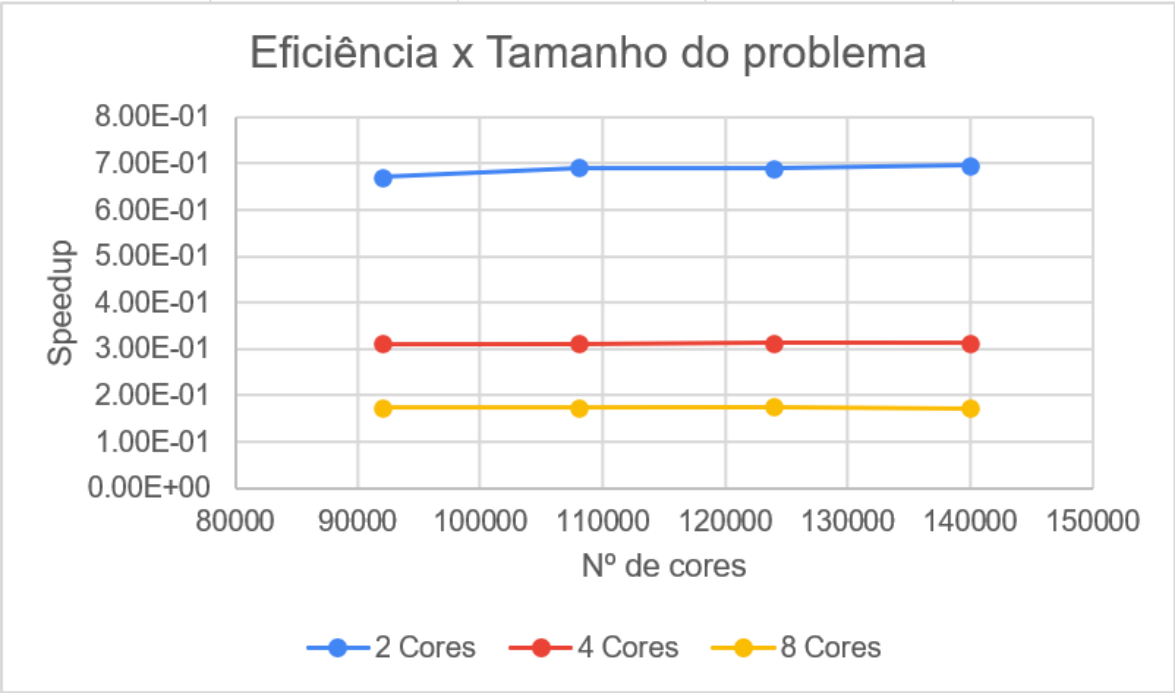
Existe um crescimento visível no speedup quando aumentamos o número de cores em execução. Além disso, observe também que o speedup aumenta proporcionalmente ao tamanho do problema. No entanto, existe uma maior variação no speedup relativo aos tamanhos de problema na execução com 8 cores, isso acontece devidos aos limites da máquina de testes, a virtualização de cores termina não entregando o mesmo desempenho que um core físico consegue demonstrar.

A tabela abaixo apresenta uma comparação mais detalhada do speedup entre o código paralelo inicial e o código paralelo otimizado, para cada tamanho de problema, por número de cores executados.

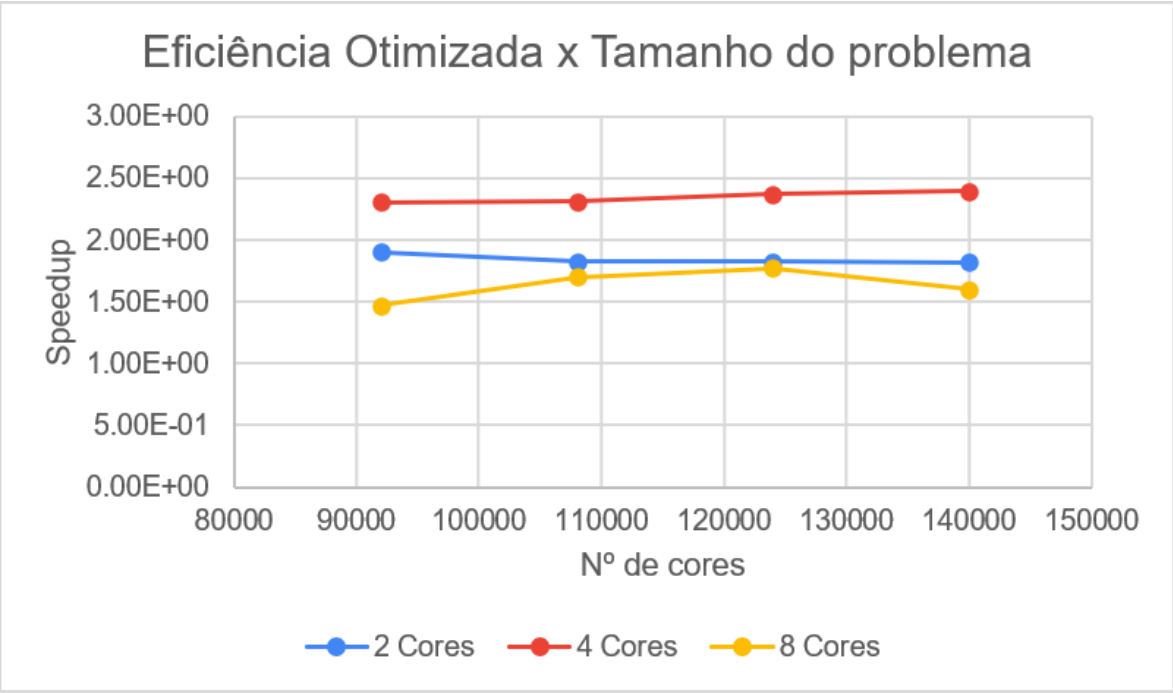
Cores	Tamanho do Problema	Speedup Inicial	Speedup Otimizado
2	92000	1.34	3.80
2	108000	1.38	3.65
2	124000	1.38	3.65
2	140000	1.39	3.64
4	92000	1.25	9.23
4	108000	1.24	9.26
4	124000	1.25	9.48
4	140000	1.25	9.57
8	92000	1.38	1.18
8	108000	1.39	1.36
8	124000	1.40	1.42
8	140000	1.38	1.28

Análise de Eficiência

Através do cálculo do speedup, é possível obter a eficiência do algoritmo quando submetido a execução com as diferentes quantidades de cores. Este cálculo pode ser realizado através da divisão do speedup do algoritmo utilizando n cores pelo número n de cores utilizados. Seria interessante executar ambos os códigos paralelo inicial e paralelo otimizado para um maior número de cores, com o objetivo de definir a escalabilidade dos algoritmos de maneira mais precisa. No entanto, como a máquina de testes possui apenas 2 cores físicos e implementa hyper-threading para executar programas em pelo menos 4 cores, a escalabilidade dos algoritmos será definida a partir da execução dos problemas com as 3 quantidades de cores definidas. Porém, vale salientar que a eficiência de cores virtuais equivale a cerca de 30% da eficiência de cores físicos.



Observando as linhas que representam a eficiência para 2, 4 e 8 cores, apesar da implementação do hyper-threading, é possível identificar uma redução da eficiência para um mesmo tamanho de problema conforme aumentamos somente o número de cores. Porém existem uma correção na eficiência conforme aumentando o tamanho do problema na mesma proporção em que aumentado o número de cores. Desse modo, é possível definir o algoritmo paralelo inicial como **fracamente escalável**.



Diferente do gráfico anterior, neste gráfico não é possível identificar com precisão o comportamento da eficiência do código paralelo otimizado conforme aumenta-se o número de cores. Seria interessante executar os mesmos problemas dessa análise para um maior número de cores em uma máquina com capacidade superior, para que as linhas assumissem características mais marcantes. No entanto, dadas as circunstâncias relativas aos limites da máquina de testes, se avaliadas somente as eficiências da execução para um mesmo tamanho de problema com 2 e 4 cores, é possível notar que existe uma correção positiva eficiência do código conforme aumentado o número de cores em utilização para todos os tamanhos de problemas. Resta deduzir que o código paralelo otimizado tende a ser **fortemente escalável**.

A tabela abaixo apresenta a eficiência calculada através dos valores de speedup anteriormente fornecidos.

Cores	Tamanho do Problema	Eficiência Inicial	Eficiência Otimizada
2	92000	0.67	1.90
2	108000	0.69	1.83
2	124000	0.69	1.83
2	140000	0.70	1.82
4	92000	0.31	2.31
4	108000	0.31	2.31
4	124000	0.31	2.37
4	140000	0.31	2.39
8	92000	0.17	1.47
8	108000	0.17	1.70
8	124000	0.17	1.77
8	140000	0.17	1.60

Obs.: Os valores superlineares encontrados para eficiência do algoritmo paralelo otimizado podem ser explicados pela otimização realizada no processo de comunicação já explicado anteriormente.

Conclusão

Considerações Finais

Foi possível obter resultados bastante satisfatórios quanto a comparação do algoritmo paralelo antes da otimização e após a otimização. É importante perceber o impacto de pequenas tarefas na rotina de comunicação entre os processos. No caso do código paralelo inicial, a realização de uma segunda ordenação a cada troca de informações entre os processos acabou diminuindo o speedup de maneira inversamente proporcional ao aumento do número de cores utilizados. Após a otimização do algoritmo paralelo o número de comparações que precisam ser realizadas a cada comunicação diminuiu. Como consequência houve uma redução no tempo de execução para cada tamanho de problema e um aumento do speedup de maneira proporcional ao aumento do número de cores em utilização. Além disso, a razão que define a eficiência também foi impactada de maneira positiva, gerando inclusive resultados superlineares. Devido aos limites da máquina de testes não foi possível definir com precisão as características do código paralelo otimizado quanto a eficiência. Porém, através dos resultados encontrados é possível inferir que o código paralelo se torna fortemente escalável após a otimização.

Softwares utilizados

```
~$: g++ --version
g++ (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0
Copyright (C) 2019 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

```
~$: python3 --version
Python 3.6.4
```

```
~$: grip --version
Grip 4.2.0
```