G2: Fases Paralelas com Otimização

Vinícius Azevedo dos Santos (13201941) – vinicius.azevedo@edu.pucrs.br

1. Introdução

Este relatório apresenta o trabalho de grau 2 da disciplina de Programação Paralela e Distribuída, as mudanças e otimizações feitas neste trabalho em relação ao projeto original foram todas individuais. Todo o código fonte deste trabalho está presente no repositório *github.com/ViniciusAz/fases-paralelas-otimizada*.

2. Implementação

Inicialmente o programa incia criando os processos, então verifica se é o processo raiz e popula um array de 600 mil números inteiros com números desordenados, desta vez, ao invés de seguir a ideia dos outros trabalhos onde é preenchido com números que representam o pior caso possível, aqui os números são gerados aleatoriamente utilizando a função rand().

Em seguida, o processo raiz faz um envio de cada parte deste array para todos os outros processos, pois cada processo fica responsável por uma única parte, para ordenar individualmente e trocar elementos com seus vizinhos caso necessário. Então inicia o laço principal do programa seguindo a ideia do pseudo-código disponibilizado no moodle. Também é possível observar que o programa está dividido em 3 fases: ordenação, comparação com os elementos vizinhos e troca de elementos com os vizinhos.

2.1. Ordenação

Na versão A do programa, toda vez que o laço volta a origem, acontece uma ordenação através do método bubble sort, para cada processo organizar os elementos do seu próprio arranjo. Já para a versão B, o algoritmo de bubblesort só acontece uma vez, no inicio da execução, e então a ordenação passa a acontecer dentro da fase de trocas.

2.2. Comparação

Cada processo é responsável por ordenar sua parte primeiro e com ela já ordenada, entramos na etapa de comparação onde, com exceção do último processo, todos mandam um *MPI_Send* com o maior valor de sua parte ao próximo vizinho que é responsável pela continuação do arranjo.

Todos os processos (exceto o raiz) receberam um valor do seu vizinho adjacente, então é feito uma comparação entre o número recebido e o primeiro número de sua parte do array, para verificar se a sequencia está ordenada em relação ao vizinho. Um feedback dessa comparação é enviado como um broadcast para todos os outros processos.

Cada processo vai ficar ciente sobre os outros processos estarem ordenados em relação aos seus vizinhos através dos broadcasts enviados e recebidos. Essas informações ficam salvas em um array que serve como flag, por exemplo: quando o processo 3 precisa trocar, no terceiro slot do arranjo de trocas, vai estar com a flag ativa. Desta forma conseguimos chegar na fase de trocas sabendo se algum processo precisa de troca e reordenação. Uma vez que ninguém precise trocar, significa que todo o array está ordenado e então a execução do sistema encerra.

2.3. Trocas

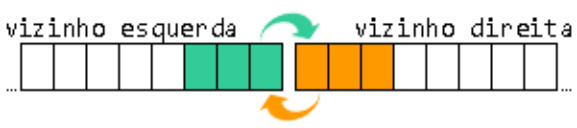

Figura 1. Exemplo Visual da Troca dos Pedaços dos Vetores.

Para as trocas na Versão A do algoritmo, o vizinho da direita copia a área que irá trocar em um novo array auxiliar, envia ao vizinho da esquerda e fica esperando uma resposta. O vizinho da esquerda recebe os valores e faz começa a fazer uma cópia de seus maiores elementos para enviar ao vizinho da direita, porém foi adicionado uma mecânica de interleaving no meio deste processo que pega ambos os arrays e ordena para enviar aos seus respectivos destinatários somente os menores ou os maiores elementos desta troca, como podemos ver na Figura 2. Então esse processo (vizinho esquerda) atualiza seus maiores valores e envia o resto dos elementos ao vizinho da direita para que ele também atualize os seus elementos.

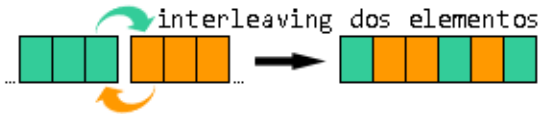

Figura 2. Troca de elementos são ordenadas com interleaving.

Na Versão B do algoritmo, não é realizada bubble sort a cada passada pelo laço de execução, porém foi adicionado a função de interleaving na hora de juntar os valores no final da troca. Por exemplo, dois processos vizinhos trocam 30% de seus valores entre si, acontecerá um interleaving desses elementos e o ambos os processos irão receber a maior e menor fatia respectivamente. O ultimo passo é integrar essa fatia ao array original, porém nessa versão, será realizado um novo interleaving distribuindo os valores em suas respectivas posições, conforme é ilustrado na figura 3.

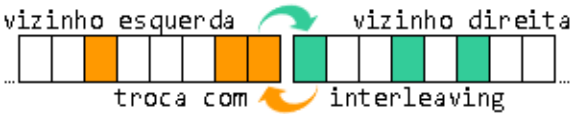

Figura 3. Execução troca Versão B conta com interleaving.

3. Avaliação

O algoritmo sequencial demorou por volta de 30 minutos, já a versão A executou em média com 16p em 97.8s e com 32p em 72.6s, já a versão B executou em média de 10.8s para 16p e 4s para 32p e por fim a melhor configuração de execução da Versão C que encontrei foi com os processos trocando 50% do seu array com cada vizinho e isso executou em média de 7.7s para 16p e 3.8s para 32p. No fim do documento há anexos contendo registros das execuções utilizadas nestes cálculos.

Em relação ao desempenho das versões, como podemos ver nos gráficos ilustrados na figura 4, houve um ganho significativo de performance em relação as Ver. A e B, o processo de NubbleSort é sempre muito lento e acaba atrasando o processamento, mesmo quando a maior parte do array já está ordenada, por isso a ver. B que faz interleaving ordenando o array na troca tem um ganho considerável, ainda mais se contar com 32p. Em relação a ver. C, diminuindo o pedaço de troca dos arrays aumentou o tempo de execução, mas aumentando a fatia melhora o tempo apesar de não ser tão expressivo quanto ao comparar as versões A e B.

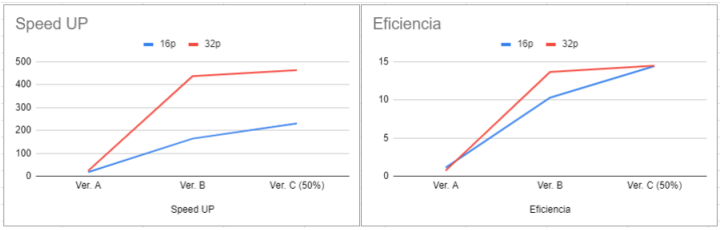

Figura 4. Gráficos de SpeedUp e Eficiência.

```c
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include "mpi.h"

void bs(int n, int * vetor)
{
    int c=0, d, troca, trocou =1;

    while (c < (n-1) & trocou )
        {
        trocou = 0;
        for (d = 0 ; d < n - c - 1; d++)
            if (vetor[d] > vetor[d+1])
                {
                troca      = vetor[d];
                vetor[d]   = vetor[d+1];
                vetor[d+1] = troca;
                trocou = 1;
                }
        c++;
        }
}

main(int argc, char** argv)
  {
  /* Parametros */
  int pedaco_troca = 30;       // 1/% do pedaco que deve ser trocado com o vizinho
  int tam_vet      = 600000;   // Tamanho total do Vetor a ser Ordenado
  int rand_range   = 1000000;  // intervalo de numeros que serao randomizados, ex: 100 = entre 0 e
     99
  int debug        = 0;        // prints debug
  int debug_limite = 0;        // ativa limite iteracoes (para entender caso entre em loop -
     problema que estava tendo com rand)
  int limite = 20;             // numero de limite de iteracoes que o programa vai rodar, caso
     debug_limite = 1
  int so_faz_um_bs = 1;        // se = 1, entao so far  BS uma vez
  int ja_fez_bs = 0;           // nao alterar

  /* Variaveis */
  int my_rank;          // Identificador deste processo
  int np;               // Numero de processos disparados pelo usuario na linha de comando (np)
  int vetor[tam_vet];   // Vetor
  int i,j,k,l;          // Variavel Auxiliar para For
  int tam_vet_pedaco;   // Tamanho do Vetor para Cada Pedaco
  int ultimo;           // Variavel Auxiliar para Ultima Posicao do Vetor do Vizinho
  int ordenado;         // Flag que verifica se est  ordenado!
  int result_troca;     // Resultado da soma do Vetor de Troca
  MPI_Status status;    // estrutura que guarda o estado de retorno
  clock_t total;        // calculo de tempo de execucao
  clock_t t_comp, comp; // calculo de tempo de comparacao
  clock_t t_troc, troc; // calculo de tempo de trocas


  MPI_Init(&argc , &argv);                // funcao que inicializa o MPI, todo o codigo paralelo
     estah abaixo
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank); // pega pega o numero do processo atual (rank)
  MPI_Comm_size(MPI_COMM_WORLD, &np);      // pega informacao do numero de processos (quantidade
     total)

    ordenado = 0;
    tam_vet_pedaco = tam_vet/np;                    // Tamanho do Pedaco que cada Processo vai
       ordenar
    int vet_aux[tam_vet_pedaco];                    // Vetor auxiliar que realiza a ordenacao
    int troca[np];                                  // Flag para saber se precisa trocar pedacos do
        vetor
    int aux[(tam_vet_pedaco*pedaco_troca/100)];     // Vetor Auxiliar para Envio das Trocas
    int aux1[(tam_vet_pedaco*pedaco_troca/100)];    // Vetor Auxiliar para Envio das Trocas
    int aux2[(tam_vet_pedaco*pedaco_troca/100 * 2)]; // Vetor para fazer o interleaving
    int i1, i2, i_aux, tam;                          // variaveis auxiliares para fazer o
       interleaving
    //printf("Tam Metade = %d", tam_vet_pedaco);
```

```
  srand(time(NULL));

/*-*-*-*-*-*-*-*-*-*-* Inicio do Processo , o Rank 0 esta coordenando -*-*-*-*-*-*-*-*-*-* */
if ( my_rank == 0 ){ // sou o primeiro?
  for(i = 0; i < tam_vet; i++){
      //vetor[i] = tam_vet - i;
      vetor[i] = rand() % rand_range;
  }
  // Manda para todo mundo o vetor
  k = tam_vet_pedaco;
  l = 0;
  for(i = 1; i < np; i++){
      l = 0;
      for(j = k; j < (k + tam_vet_pedaco); j++){
          vet_aux[l] = vetor[j];
          l++;
      }
      k = k + tam_vet_pedaco;
      MPI_Send(&vet_aux, tam_vet_pedaco, MPI_INT, i, 1, MPI_COMM_WORLD);
  }
}else
  MPI_Recv(&vet_aux, tam_vet_pedaco, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status); // recebo
      primeiro vetor

  // Arrumar o vetor do Rank 0
  if(my_rank == 0)
      for(i = 0; i < tam_vet_pedaco; i++){
          vet_aux[i] = vetor[i];
      }

/* -*-*-*-*-*-*-*-*-*-* Fim do Processo inicial -*-*-*-*-*-*-*-*-*-* */

  total = clock();
  t_troc = 0;
  t_comp = 0;

  while(ordenado == 0){
      if(so_faz_um_bs == 0 || (so_faz_um_bs == 1 && ja_fez_bs == 0)) {
          bs(tam_vet_pedaco,vet_aux);
          ja_fez_bs = 1;
      }


      comp = clock();
      troca[my_rank] = 0;                          // Informa que a troca nao e necessaria

      // print  VETOR ordenado do PROCESSO
      if(debug == 1) { printf(" \n BS PID : %d = ", my_rank); for(i = 0; i < tam_vet_pedaco; i++)
          printf(" %d, ", vet_aux[i]); printf(" \n"); }


      if(my_rank != (np-1))                        // Se eu nao for o ultimo eu mando para meu
          vizinho da direita
          MPI_Send(&vet_aux[tam_vet_pedaco - 1], 1, MPI_INT, (my_rank+1), 1, MPI_COMM_WORLD);
      if(my_rank != 0){
          MPI_Recv(&ultimo, 1, MPI_INT, (my_rank -1), MPI_ANY_TAG, MPI_COMM_WORLD, &status);
          if(ultimo > vet_aux[0]){ // Precisa trocar
              troca[my_rank] = 1;
              if(debug == 1) { printf("\n Sou o PID %d e o ult    [%d] > {%d} primeiro e troca = %
                  d  ",my_rank ,ultimo, vet_aux[0], troca[my_rank]); }
          }else{
              if(debug == 1) { printf("\n Sou o PID %d e o ult    [%d] > {%d} primeiro e troca = %
                  d  ",my_rank ,ultimo, vet_aux[0], troca[my_rank]); } // Nao precisa Trocar
              troca[my_rank] = 0;
          }
      }

      // Multicast
      for(i = 0; i < np; i++)
          MPI_Bcast(&troca[i], 1, MPI_INT, i, MPI_COMM_WORLD);
      MPI_Barrier(MPI_COMM_WORLD);
```

```
        // Imprimo o Vetor de Troca
        result_troca = 0;
        for(i = 0; i < np; i++){
            //printf("\n Sou o %d e o valor do TROCA = %d ",my_rank ,troca[i]);
            result_troca = result_troca + troca[i];
        }
        if(debug == 1) { if(my_rank == 0) printf("\n result troca = %d", result_troca); }
        if(result_troca > 0){
            ordenado = 0;
            //printf("\n VOU ORDENAR MAIS!%d", ordenado);

        }else{
            ordenado = 1;
            //printf("\n ACABEI %d", ordenado);
        }

        MPI_Barrier(MPI_COMM_WORLD);

        t_comp += clock() - comp;

        /* Processo de Troca */
        troc = clock();

        if(troca[my_rank] == 1 && my_rank > 0){
            //printf(" \n DEBUG Proc1 Troca Pid: %d : ", my_rank);
            for(i = 0; i < (tam_vet_pedaco*pedaco_troca/100);i++){
                aux[i] = vet_aux[i];
            }
            MPI_Send(&aux, (tam_vet_pedaco*pedaco_troca/100), MPI_INT, (my_rank-1), 1,
                MPI_COMM_WORLD);
            MPI_Recv(&aux1, (tam_vet_pedaco*pedaco_troca/100), MPI_INT, (my_rank-1), MPI_ANY_TAG,
                MPI_COMM_WORLD, &status);
            //for(i = 0; i < (tam_vet_pedaco*pedaco_troca/100);i++){
            //    vet_aux[i] = aux1[i];
            //}

            // interleaving ordenando novos valores
            tam = (tam_vet_pedaco*pedaco_troca/100);
            i1 = tam;
            i2 = 0;

            for (i_aux = 0; i_aux < tam_vet_pedaco; i_aux++) {
                if (((vet_aux[i1] <= aux1[i2]) && (i1 < tam_vet_pedaco)) || (i2 == tam))
                    aux2[i_aux] = vet_aux[i1++];
                else
                    aux2[i_aux] = aux1[i2++];
            }
            for(i = 0; i < tam_vet_pedaco; i++) {
                vet_aux[i] = aux2[i];
            }
            //fim interleaving
        }

        if(troca[my_rank + 1] == 1 && my_rank < (np-1)){
            //printf(" \n DEBUG Proc2 Troca Pid: %d : ", my_rank);
            MPI_Recv(&aux, (tam_vet_pedaco*pedaco_troca/100), MPI_INT, (my_rank+1), MPI_ANY_TAG,
                MPI_COMM_WORLD, &status);
            k = 0;
            for(i = tam_vet_pedaco - (tam_vet_pedaco*pedaco_troca/100); i < tam_vet_pedaco;i++){
                aux1[k] = vet_aux[i];
                k++;
            }

            //interleaving
            tam = (tam_vet_pedaco*pedaco_troca/100 * 2); //pedaco final + inicial que dois processos
                 estao trocando
            i1 = 0;
            i2 = 0;

            for (i_aux = 0; i_aux < tam; i_aux++) {
                if (((aux[i1] <= aux1[i2]) && (i1 < (tam / 2))) || (i2 == (tam / 2)))
                    aux2[i_aux] = aux[i1++];
                else
```

```
                    aux2[i_aux] = aux1[i2++];
            }

            if(debug == 1) { printf(" \n Interleaving Pid: %d : ", my_rank); for(i = 0; i < tam; i
                ++){ printf(" %d, ", aux2[i]); } printf("\n "); }

            for(i = 0; i < tam/2; i++) {
                aux[i] = aux2[i];
                aux1[i] = aux2[i+(tam/2)];
            }
            //fim interleaving

            //k = 0;
            //for(i = tam_vet_pedaco - (tam_vet_pedaco*pedaco_troca/100); i < tam_vet_pedaco;i++){
            //    vet_aux[i] = aux[k];
            //    k++;
           // }

            // interleaving ordenando novos valores

            tam = (tam_vet_pedaco*pedaco_troca/100);
            i1 = 0;
            i2 = 0;

            for (i_aux = 0; i_aux < tam_vet_pedaco; i_aux++) {
                if (((vet_aux[i1] <= aux[i2]) && (i1 < (tam_vet_pedaco - tam))) || (i2 == tam))
                    aux2[i_aux] = vet_aux[i1++];
                else
                    aux2[i_aux] = aux[i2++];
            }
            for(i = 0; i < tam_vet_pedaco; i++) {
                vet_aux[i] = aux2[i];
            }
            //fim interleaving

            MPI_Send(&aux1, (tam_vet_pedaco*pedaco_troca/100), MPI_INT, (my_rank+1), 1,
                MPI_COMM_WORLD);
        }


        MPI_Barrier(MPI_COMM_WORLD);
        t_troc += clock() - troc;


        if(debug_limite == 1) { if(limite == 0) ordenado = 1; limite--; }
    }


    /* DEBUG VETOR FINAL do PROCESSO */
    if(debug == 1) { printf(" \n VETOR FINAL Pid: %d : ", my_rank); for(i = 0; i < tam_vet_pedaco; i
        ++){ printf(" %d, ", vet_aux[i]); } printf("\n "); }

    total = clock() - total;
    double time_taken = ((double)total)/CLOCKS_PER_SEC; // in seconds
    double time_troc = ((double)t_troc)/CLOCKS_PER_SEC; // in seconds
    double time_comp = ((double)t_comp)/CLOCKS_PER_SEC; // in seconds
    double troc_percent = (time_troc * 100) / time_taken;
    double comp_percent = (time_comp * 100) / time_taken;
    if(my_rank == 0) printf("Tempo total = %.2f segundos para ordenar %d mil elementos (-np %d) \n",
         time_taken, (tam_vet/1000), np);
    if(my_rank == 0) printf("Compara  o = %.1f segundos (%.2f%s) - Troca = %.1f segundos (%.2f%s)
        \n", time_comp, comp_percent, "%", time_troc, troc_percent, "%");


  MPI_Finalize();
}
```

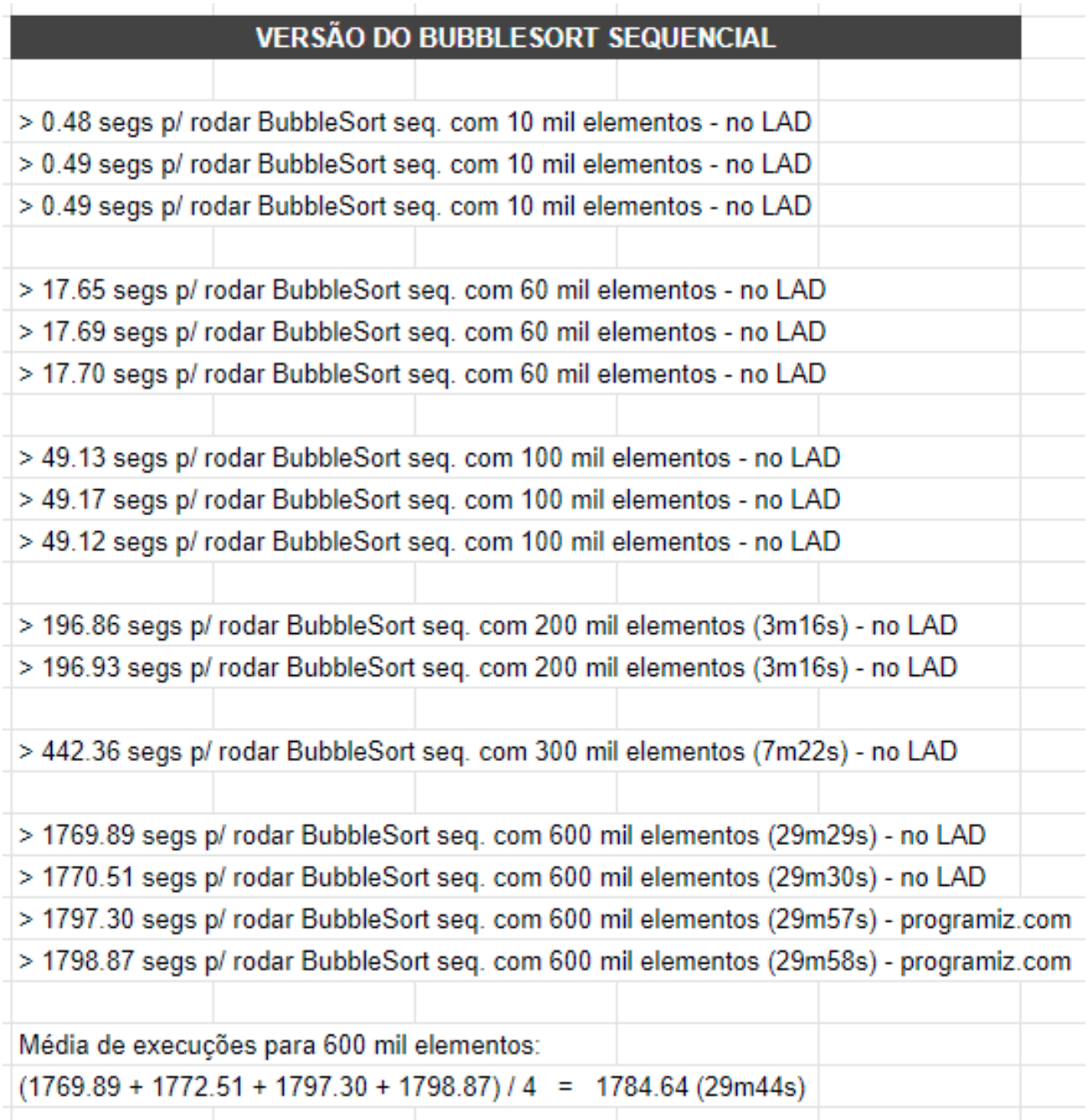

Figura 5. Dados obtidos executando versão sequencial do BS.

VERSAO A, SEMPRE FAZ O BUBBLESORT, FAZ INTERLEAVING NAS TROCAS
ladrun -np 32 tf
Tempo total = 71.19 segundos para ordenar 600 mil elementos (-np 32)
Comparação = 22.8 segundos (32.08%) - Troca = 0.6 segundos (0.79%)
Tempo total = 74.53 segundos para ordenar 600 mil elementos (-np 32)
Comparação = 25.7 segundos (34.52%) - Troca = 0.6 segundos (0.75%)
Tempo total = 72.00 segundos para ordenar 600 mil elementos (-np 32)
Comparação = 23.9 segundos (33.26%) - Troca = 0.6 segundos (0.86%)
Média -np 32 Total : 72.57
Média -np 32 Comparação : 24.13
Média -np 32 Trocas : 0.6
ladrun -np 16 tf
Tempo total = 81.39 segundos para ordenar 600 mil elementos (-np 16)
Comparação = 23.1 segundos (28.35%) - Troca = 0.2 segundos (0.29%)
Tempo total = 104.24 segundos para ordenar 600 mil elementos (-np 16)
Comparação = 29.6 segundos (28.35%) - Troca = 0.2 segundos (0.22%)
Tempo total = 104.97 segundos para ordenar 600 mil elementos (-np 16)
Comparação = 29.7 segundos (28.30%) - Troca = 0.2 segundos (0.24%)
Média -np 16 Total : 97.86
Média -np 16 Comparação : 27.46
Média -np 16 Trocas : 0.2

Figura 6. Dados obtidos executando versão A.

VERSAO B, FAZ O BUBBLESORT UMA VEZ, FAZ INTERLEAVING
ladrun -np 32 tf
Tempo total = 4.06 segundos para ordenar 600 mil elementos (-np 32)
Comparação = 0.1 segundos (1.97%) - Troca = 1.1 segundos (26.35%)
Tempo total = 4.15 segundos para ordenar 600 mil elementos (-np 32)
Comparação = 0.1 segundos (3.13%) - Troca = 1.1 segundos (26.51%)
Tempo total = 4.07 segundos para ordenar 600 mil elementos (-np 32)
Comparação = 0.1 segundos (1.47%) - Troca = 1.1 segundos (26.78%)
Média -np 32 Total : 4.09
Média -np 32 Comparação : 0.1
Média -np 32 Trocas : 1.1
ladrun -np 16 tf
Tempo total = 10.86 segundos para ordenar 600 mil elementos (-np 16)
Comparação = 0.0 segundos (0.37%) - Troca = 0.6 segundos (5.89%)
Tempo total = 10.85 segundos para ordenar 600 mil elementos (-np 16)
Comparação = 0.1 segundos (1.11%) - Troca = 0.7 segundos (6.27%)
Tempo total = 10.86 segundos para ordenar 600 mil elementos (-np 16)
Comparação = 0.1 segundos (0.74%) - Troca = 0.7 segundos (6.26%)
Média -np 16 Total : 10.86
Média -np 16 Comparação : 0.1
Média -np 16 Trocas : 0.7

Figura 7. Dados obtidos executando versão B.

VERSAO C, IGUAL B, VARIANDO PERCENTUAL DE TROCA
PEDAÇO DE TROCA DE 50% ladrun -np 32 tf
Tempo total = 3.87 segundos para ordenar 600 mil elementos (-np 32)
Comparação = 0.1 segundos (2.84%) - Troca = 0.8 segundos (21.71%)
Tempo total = 3.86 segundos para ordenar 600 mil elementos (-np 32)
Comparação = 0.0 segundos (1.04%) - Troca = 0.9 segundos (23.06%)
Tempo total = 3.86 segundos para ordenar 600 mil elementos (-np 32)
Comparação = 0.1 segundos (1.55%) - Troca = 0.9 segundos (22.80%)
PEDAÇO DE TROCA DE 40% ladrun -np 32 tf
Tempo total = 4.06 segundos para ordenar 600 mil elementos (-np 32)
Comparação = 0.1 segundos (1.48%) - Troca = 1.1 segundos (26.60%)
Tempo total = 4.14 segundos para ordenar 600 mil elementos (-np 32)
Comparação = 0.1 segundos (2.17%) - Troca = 1.1 segundos (27.54%)
Tempo total = 4.11 segundos para ordenar 600 mil elementos (-np 32)
Comparação = 0.1 segundos (1.70%) - Troca = 1.1 segundos (27.01%)
PEDAÇO DE TROCA DE 20% ladrun -np 32 tf
Tempo total = 4.10 segundos para ordenar 600 mil elementos (-np 32)
Comparação = 0.1 segundos (1.71%) - Troca = 1.1 segundos (27.07%)
Tempo total = 4.13 segundos para ordenar 600 mil elementos (-np 32)
Comparação = 0.1 segundos (1.69%) - Troca = 1.1 segundos (27.60%)
Tempo total = 4.09 segundos para ordenar 600 mil elementos (-np 32)
Comparação = 0.1 segundos (1.22%) - Troca = 1.1 segundos (27.63%)

Figura 8. Dados obtidos executando versão C.

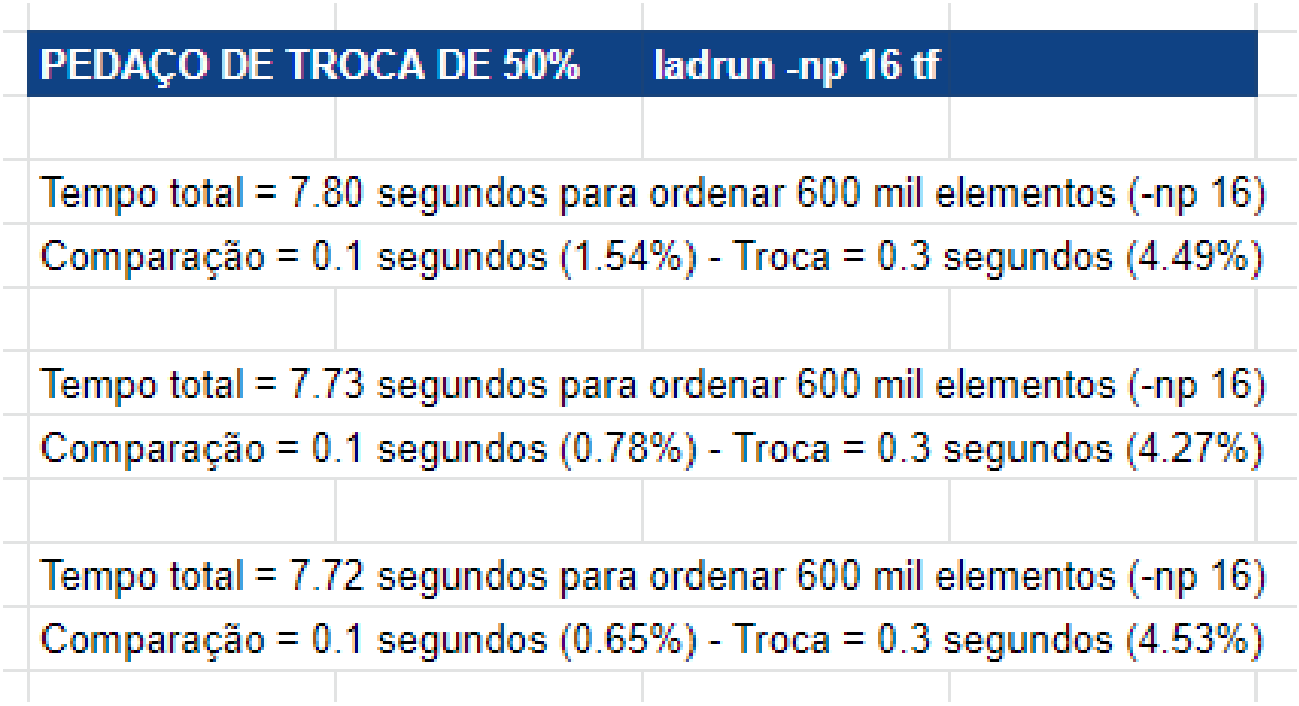

Figura 9. Dados obtidos executando versão C com np 16.