Instituto federal de educação ciência e tecnologia.

Curso: Tecnologia em análise e desenvolvimento de sistemas.

Autor: Lucas Alves da Costa.

Matricula: 20151070130183.

Formosa-GO - 24/01/2018

Trabalho Final de Estrutura de Dados I.

Ordenação e busca utilizando os algoritmos (*Quicksort e MergeSort*).

Introdução.

O Quicksort e o MergeSort são algoritmos bem renomados no mundo da tecnologia, sua principal característica é dividir o problema em partes para resolve-lo com o máximo de eficiência possível, este relatório irá focar na eficiência de cada um deles.

Para isso adotarei uma forma de fazer com que esses algoritmos trabalhem em paralelo usando um método chamado thread, esse método consiste em criar uma linha de execução para uma determinada funcionalidade em uma parte do processador, sendo possível criar outros processos para serem executados em outras partes (núcleos) do processador.

Após implementar cada um destes algoritmos para operarem utilizando threads, irei implementar um sistema de busca que irá buscar um determinado valor presente em um vetor que foi ordenado por um destes métodos de ordenação e farei novamente a busca no vetor que foi ordenado pelo outro método de ordenação.

Para saber qual é mais eficiente, irei medir o tempo que cada um gasta para ordenar um determinado vetor de números aleatórios, e em seguida irei usar o sistema de busca para retornar um valor presente em cada vetor recém ordenado. Só será possível fazer a busca de um elemento se o vetor estiver totalmente ordenado, então o algoritmo que apresentar uma busca mais rápida será por consequência o algoritmo mais rápido na ordenação dos valores.

Algoritmo utilizado para resolução do problema.

Problema proposto; criar um sistema de busca utilizando os métodos *Quicksort* e *MergeSort*, usando a técnica de Treads (Biblioteca de ptreads da linguagem C). E no final da busca feita utilizando os dois métodos mostrar qual foi mais rápido.

A heurística utilizada para resolver o problema proposto foi; usar um thread para executar o algoritmo Quicksort, esse algoritmo usará o sistema de recursividade para reexecutar o mesmo algoritmo várias outras vezes com parâmetros de entrada diferentes, para cada chamada recursiva será criado um novo thread para a função chamada. Após o processo de ordenação irei usar o sistema de busca para retornar um determinado valor.

Em seguida usarei o mesmo conceito de thread para o MergeSort, ou seja, esse algoritmo também irá ser executado em uma linha de processamento independente, como o MergeSort trabalha com o sistema recursivo, o mesmo irá reexecutar suas funções várias vezes até que o problema seja resolvido, para cada sub-rotina será criada um novo thread independente para que a mesma seja executada. Após esse processo de ordenação concluído, irei fazer outra busca pelo mesmo valor, só que desta vez irei procurar no vetor ordenado pelo MergeSort.

O sistema de busca utilizado foi o de busca binária, onde é necessário que os vetores estejam previamente ordenados, esse sistema caiu muito bem neste contexto pois, já que o foco do quicksort e do mergesort é a ordenação, então o algoritmo que ordenar mais rápido será o que possibilitar a busca mais rápida.

Para medir o tempo de execução usei uma função de tempo onde calculei o intervalo entre o tempo de início e o tempo final da execução de cada algoritmo, sendo que o algoritmo de busca será calculado juntamente com cada um dos algoritmos de ordenação, no final será possível saber qual dos dois algoritmos foi mais rápido na ordenação, juntamente com a busca.

Essa forma de resolução do problema foi implementada da seguinte forma:

```
#include <stdio.h>

#include <stdlib.h>

#include <conio.h>

#include <time.h>// biblioteca onde se encontram as funções de tempo utilizadas neste
projeto.

#include <pthread.h>// biblioteca onde se encontram as funções necessárias para criar as
threads.

#define size_vet  1000// tamanho máximo dos vetores do quick e do merge.


        int vetorQuick[size_vet];

        int vetorMerge[size_vet];

        int send;

        clock_t inicio, fim;
```

```
	clock_t inicio2, fim2;


	double tempo,tempo2;

	// são criados dois vetores de tamanho fixo para gardar valores aleatórios e posteriormente serem ordenados.

typedef struct NO_q{

	int dir;

	int esq;

}no_q;


typedef struct node {

    int i;

    int j;

} NODE;

// estruturas para guardar valores de argumentos para funções que vão operar em base de threads.


void *q_sort(void *parametro){

		printf("*");// será imprimido este operador sempre que a função estiver em operação.

		pthread_t td1,td2;


	 no_q *param;

		param = (no_q*)parametro;// convertendo o parâmetro da função para facilitar o trabalho.


		no_q *no1 = (no_q*)malloc(sizeof(no_q));

		no_q *no2 = (no_q*)malloc(sizeof(no_q));

		int *vetor = vetorQuick;

		   int esquerda = param->esq;

		   int direita = param->dir;

		   int tamanho = direita;

	int i, j, meio, y;

	int execute;


	i= esquerda;// 'i' representa o último valor a esquerda do vetor
```

```
       j= direita;  // 'j' representa o primeiro valor a direita do vetor


       meio = vetor[(esquerda+direita)/2]; // 'meio' recebe a posição central"MEIO" do
vetor

        while(i<=j){// enquanto a direita for maior que a esquerda


              while(vetor[i] < meio && i < direita){// enquanto o vetor na primeira posicao
a esquerda for MENOR q o meio do vetor E o último elemento a direita

                                                    // do vetor for maior que o primeiro a
esquerda (que vai até o meio) eu avanço a direita 'i'

                     i++;

                     }

                       while(vetor[j] > meio && j > esquerda){

                                                          j--;

                       }// enquanto o fim do vetor a esquerda for MAIOR que o meio do vetor
E a variavel 'J -> última posição for maior que a esquerda(que vai até no meio do vetor)


         if(i <= j){  // se a esquerda atual for menor OU igual a direita atual faço as
trocas


                             y= vetor[i];  // recebo essa posição da esquerda

 vetor[i]=vetor[j]; // troco com a posição do vetor da direita

vetor[j]=y;  // agora a cópia do valor que estava na posição do vetor da 'direita atual'
vai para a esquerda atual

                              i++;   // avanço para o meio do vetor

                              j--;   // retrocesso da posição até o meio do vetor

                        }

              }


        if(j > esquerda){ // parte recursiva onde eu verifico se a direita ATUAL é maior
que a esquerda passa por parâmetro


         no1->esq = esquerda;

         no1->dir = j;

    execute = pthread_create(&td1, NULL, q_sort, (void *)no1);// executo a primeira thread

             if (execute){

               printf("%d %s - unable to create thread - ret - %d\n", __LINE__,
__FUNCTION__, execute);
```

```
                 exit(1);
                }
               }

         if(i < direita){ // se a  esquerda atual for maior que a direita passada por
parametro

                no2->esq = i;
                no2->dir = direita;
      execute = pthread_create(&td2, NULL, q_sort, (void *)no2);// executo o segundo thread
                if (execute){
                printf("%d %s - unable to create thread - ret - %d\n", __LINE__,
__FUNCTION__, execute);
                exit(1);
               }
               pthread_join(td1,NULL);
               pthread_join(td2,NULL);
        // a parte do JOIN faz com que cada thread seja executado em alguma parte do
processador, neste caso cada thread irá operar de forma independente em uma linha de
execução do processador.
              }
              pthread_exit(NULL);
}
/*************************************************************************/
```

O Quicksort consiste em dividir um vetor desordenado em três partes, que são; o pivô, que neste caso é o elemento central do vetor, e os elementos maiores que o pivô, e os elementos menores que o pivô.

Com essa divisão feita o Quicksort irá aplicar a ordenação para primeira metade deste vetor onde se encontram os valores menores que o pivô, e em seguida, para segunda metade do vetor onde estão os valores maiores que o pivô.

Sem o uso de threads só seria possível aplicar a ordenação para um lado do vetor de cada vez, não sendo possível que uma função trabalhe em paralelo com outra quando for recursivamente chamada.

Como estou usando threads, é possível que duas funções operem em paralelo uma com outra, sendo uma ordenando os itens menores que o pivô em uma parte do processador, enquanto a

outra parte está ordenando os elementos maiores que o pivô em outra parte do processador. Como uma recursão não precisa esperar a outra ser concluída para ser executada então o tempo de processamento do algoritmo é relativamente mais rápido do que no caso normal.

Vamos analisar agora o MergeSort.

```
/*************************************************************************/

void merge(int inicio, int fim){
            int meio = (inicio+fim)/2;
            int i, j, k;
            int n1 = meio - inicio + 1;// recebe o tamanho de uma metade do array
            int n2 =  fim - meio;// recebe o tamanho da outra metade do array
            int *vetor = vetorMerge;// recebendo o vetor desordenado.

            /*Dois vetores auxiliares definidos de forma global.
          Cada um dos vetores recebe o tamanho exato de cada metade de forma dinâmica*/
            int *temp_L = (int*)malloc(n1*sizeof(int));
            int *temp_R = (int*)malloc(n2*sizeof(int));
              if(!temp_L || !temp_R){
               printf("SEM MEMORIA DISPONIVEL!\n\n");
               exit(1);
                 }
               temp_L[n1], temp_R[n2];

            /* copiando os dados da primeira parte do vetor para o vetor auxiliar*/
            for (i = 0; i < n1; i++){
               temp_L[i] = vetor[inicio + i];
               }
                /*copiando a segunda metade do vetor para o segundo vetor auxiliar*/
            for (j = 0; j < n2; j++){
                temp_R[j] = vetor[meio + 1+ j];
               }
            /* reiniciando as variaveis*/
            i = 0;
            j = 0;
            k = inicio;
```

```
/*Momento da troca de valores onde comparo se o valor de um lado do vetor é menor ou igual
ao valor do outro lado do vetor*/

            while (i < n1 && j < n2) {

                if (temp_L[i] <= temp_R[j]){

vetor[k] = temp_L[i];

// depois da comparação jogo os valores menores dentro do vetor original

                    i++;// avanço para o meio

                      }

                else

                {

                    vetor[k] = temp_R[j];// jogo sempre o valor menor dentro do vetor
original

                    j++; // avanço para o fim

                }

                k++;// sobe uma posição

            }

 /* copio o restante dos elementos "caso houver" do primeiro vetor auxiliar para o vetor
original */

            while (i < n1){

                vetor[k] = temp_L[i];

                i++;

                k++;

            }


            /* copiando valores do outro vetor auxiliar para o vetor original */

            while (j < n2){

                vetor[k] = temp_R[j];

                j++;

                k++;

            }

        free(temp_L);

        free(temp_R);

}


void *MergeSort(void *a){
```

```
          printf("+");
        NODE *p = (NODE *)a;
        NODE n1, n2;
        int mid = (p->i+p->j)/2;// calculando o meio do vetor
        pthread_t tid1, tid2;// variáveis do tipo thread
        int ret;

        n1.i = p->i;
        n1.j = mid;

        n2.i = mid+1;
        n2.j = p->j;
// parâmetros da função que será chamada recursivamente
        if (p->i >= p->j) return;

        ret = pthread_create(&tid1, NULL, MergeSort, &n1);
        if (ret) {
                printf("%d %s - unable to create thread - ret - %d\n", __LINE__,
__FUNCTION__, ret);
                exit(1);
        }
//Executando o primeiro thread para primeira chamada recursiva da função.

        ret = pthread_create(&tid2, NULL, MergeSort, &n2);
        if (ret) {
                printf("%d %s - unable to create thread - ret - %d\n", __LINE__,
__FUNCTION__, ret);
                exit(1);
        }
//Executando o segundo thread para segunda chamada recursiva da função.
        pthread_join(tid1, NULL);
        pthread_join(tid2, NULL);
//Usando o JOIN para forçar o paralelismo entre uma chamada recursiva e outra.
        merge(p->i, p->j);// função responsável por fazer a troca dos elementos entre os
vetores.
        pthread_exit(NULL);
```

```
}
```

Vamos entender o funcionamento do MergeSort implementado aqui.

O MergeSort consiste em dividir um vetor em duas partes, sendo elas do começo até o meio, e do meio +1 até o final do vetor, cada uma destas partes será subdividida em outras duas partes e em seguida será feito a troca de valores entre essas subdivisões dos vetores, o passo seguinte é unir as partes que foram separadas e reorganizadas. Isso será feito recursivamente por várias vezes até que não seja mais possível ordenar.

No caso normal a ordenação será possível apenas para um lado do vetor de cada vez, sendo que uma parte do vetor será quebrada em outras duas partes, e após isso será feita a troca e a união das partes subdivididas. O mesmo caso se aplica a outra parte do vetor, só que antes da subdivisão e troca de elementos é necessário esperar que a primeira chamada recursiva da função seja executada.

Com o uso de threads, posso criar uma linha de execução para cada função recursiva que irá trabalhar com uma parte do vetor. O algoritmo ficará mais rápido pois, se um vetor é dividido em duas partes e cada parte será subdivida em outras duas partes, então posso fazer a troca de valores em uma parte do vetor enquanto faço a troca de valores em outra parte do vetor de forma paralela. Sem a necessidade de esperar uma chamada da função ser encerrada para trabalhar com a sequente, temos então um algoritmo mais rápido.

Voamos agora a função que irá imprimir os vetores após a ordenação.

```
/*<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>*/
void imprime(int vetor[], int size){
	int i;
	printf("vetor ordenado-> ");
	for(i=0; i<=size-1; i++){
		printf("[%d] " ,vetor[i]);
	}printf("\nEND\n");
}
```

Essa função não possui comentários pois seu funcionamento é muito simples, primeiro recebemos o vetor a ser impresso e seu respectivo tamanho, segundo, um laço se encarregará de pegar todas as posições deste vetor, desde a posição zero até a última posição determinada pelo tamanho do vetor, o valor de cada posição será impresso na tela para o usuário.

Vamos ao algoritmo de busca binária.

```
//-----------------------------------------------------------------//

void seek_bin(int valor, int size_m, int *MATRIZ){


    int found = 0; // enquanto o valor da variável for zero o valor não foi encontrado.

    int high = size_m;

    int low = 0;

    int i=0;


    int middle = (high + low)/2; // calculando o meio do vetor.


     while((!found) && (high >= low)){

           if(valor == MATRIZ[middle]){

               found = 1; // caso o valor seja encontrado a variável forçará o encerramento
da busca e irá apresentar o valor encontrado no vetor e sua respectiva posição.

               printf("\nvalor <%d> encontrado na posicao[%d]\n",valor,middle);


printf("_____________________________________________________________\n");

               sleep(3); // aguardo três segundos antes de executar outro procedimento.

                 }

            else

              if(valor < MATRIZ[middle]){

// se o valor for menor que o valor do meio do vetor então iremos procurar na parte
esquerda do vetor.

                high = middle -1;

                            }

                     else{

// se o valor for maior que o elemento do meio do vetor então iremos procurar na parte
direita do vetor, e assim por diante.

                     low = middle + 1;

                            }



           middle = (high + low) /2;

           i++;

      }

      if(found ==0){
```

```
// casso o valor não seja encontrado no vetor será apresentado a seguinte mensagem;
printf("\nValor <%d> nao encontrado neste Vetor[...]\n-----------------------------------
----------------------------\n",valor);
       sleep(3);// aguardo três segundos para executar outro pocesso.
         }
}
```

Vamos entender o funcionamento da busca binaria.

Esse algoritmo funciona de forma semelhante a estrutura de arvores binárias, pois consistem em verificar o valor central do vetor previamente ordenado, e compara-lo com o elemento que está sendo procurado, caso o elemento de busca seja maior que o valor central do vetor então a busca será feita no lado direito do vetor, caso o elemento seja menor que o valor central do vetor então a busca será feita no lado esquerdo do vetor.

Nestes dois casos será descartada a parte que não contém o valor, e a verificação será feita novamente até que o elemento de busca seja igual ao valor central do vetor.

Vamos a parte de inicialização das funções principais como thread, e o preenchimento dos vetores de forma aleatória.

```
void inicializa(){
       no_q *no1 = (no_q*)malloc(sizeof(no_q));
       NODE no2;
       pthread_t t_quick, t_merge; //  variáveis do tipo thread.

       int execute;
       int i;
       int aleatorio;

        srand(time(NULL));
        for(i=0; i<=size_vet-1; i++){
              aleatorio = (rand() % 2000);
              vetorQuick[i] = aleatorio;
              vetorMerge[i] = aleatorio+1;
        }
        // Esse laço preenche os vetores com valores aleatórios gerados pela função rand();
```

```
            // para garantir que os vetores não sejam idênticos, o vetor do MergeSort terá
todos os valores gerados acrescidos de um.
        no1->esq = 0;
        no1->dir = size_vet-1;// ciando parâmetros para função Quicksort.


        no2.i = 0;
        no2.j = size_vet-1;// criando parâmetros para função MergeSort.


         inicio = clock();// disparando um cronometro para marcar o tempo de execução.
         execute = pthread_create(&t_quick, NULL, q_sort, (void *)no1);
         pthread_join(t_quick,NULL);
```

No primeiro thread, o Quicksort será executado. Como padrão, os parâmetros da função do quick são passados separadamente dentro da função; pthread_create;

```
        if (execute!=0){
                printf("%d %s - unable to create thread - ret - %d\n", __LINE__,
__FUNCTION__, execute);
                exit(1);
      }
       seek_bin(send,size_vet,vetorQuick);// realizando a busca binária.
       fim = clock();
       tempo = (fim-inicio/CLOCKS_PER_SEC);// recebo o tempo final da execução do
algoritmo.


       inicio2 = clock()-fim;// forncando o reinicio do cronometro para essa variável.
       execute = pthread_create(&t_merge, NULL, MergeSort, (void *)&no2);
          pthread_join(t_merge,NULL);


       if(execute!=0){
               printf("%d %s - unable to create thread - ret - %d\n", __LINE__,
__FUNCTION__, execute);
                exit(1);
          }
          seek_bin(send,size_vet,vetorMerge);
          fim2 = clock()-fim;// recebo o tempo corrente do cronometro menos o tempo de
execução do primeiro algoritmo.
          tempo2 = (fim2-inicio2/CLOCKS_PER_SEC);// cálculo para saber o tempo final de
execução da função.
```

```
}
```

Essa função funciona da seguinte forma, primeiro os vetores usados pelo Quicksort e o MergeSort são preenchidos por números aleatórios, mas a diferença é que o vetor do MergeSort irá ter os valores do Quicksort somados de um.

Em seguida são criadas as estruturas de parâmetros que serão usadas nas funções thread juntamente com sua função de ordenação correspondente.

Disparo em seguida um cronometro de tempo com a função *clock()*, executo o primeiro thread para o Quicksort e em seguida é executado a função de busca binária, após retornar ou não o valor procurado, recebo o tempo do cronometro no final da execução, calculo o tempo e guardo em uma variável para que esse valor possa ser usado posteriormente.

Para a função seguinte a variável de tempo recebe o tempo corrente do cronometro menos o tempo do outro algoritmo para reiniciar o valor do cronometro para aquela variável, criando por consequência um valor entre zero e 0,00001, considerando a margem de erro.

Próximo passo é executado o thread da função MergeSort, em seguida a função de busca binária retornara ou não o valor procurado, pois caso não encontrado, a função de busca irá retornar uma mensagem informando que o valor não foi encontrado no vetor correspondente.

Depois da função de busca uma variável de tempo recebe o tempo corrente do cronometro menos o tempo de execução do thread anterior. Porque isso?

Exemplo: supomos que o tempo final de execução do Quicksort acoplado no thread foi de 5 segundos, a partir daí o tempo continua correndo e a função do MergeSort com thread é executada, digamos que essa última função demorou mais quatro segundos para ser executada, totalizando 9 segundos, como queremos apenas o tempo de execução deste último procedimento então iremos apenas subtrair o tempo final do último procedimento com o tempo final do primeiro processo. Sendo então (9 menos 5) totalizando 4 segundos de execução deste último procedimento. Vamos agora a função de entrada.

```
void entrada(){
printf("\tDIGITE UM VALOR ALEATORIO ENTRE: ZER0->[0]_&_DOIS MIL->[2000]!\nVALOR->.");
	scanf("%d",&send);
	sleep(1,5);
}
```

Essa função é responsável por receber o valor que será procurado nos dois vetores. Não se confunda, ela apenas recebe o valor de busca, mas quem faz a busca é a função de busca binária. Depois de receber o elemento de busca será aguardado um segundo e meio para que a próxima função seja executada.

Passamos em fim para função principal onde tudo começa e termina.

```
int main(int argc, char *argv[]) {

            entrada();// chamo a função responsável pelo valor de busca

            inicializa(); // função de criação execução e cronometro de tempo dos threads.

           printf("\n\n\t>>>ORDENACAO CONCLUIDA!!\n\n");

           printf("\nTempo de ordenacao do QUICKSORT() = [%lf] M.segundos..\n",tempo);

       seek_bin(send,size_vet,vetorQuick);

        printf("\nTempo de ordenacao do MERGESORT() = [%lf] M.segundos..\n",tempo2);

            seek_bin(send,size_vet,vetorMerge);

            printf("A diferenca de tempo entre O Quick & Merge foi [%lf]
M.segundos...\n",difftime(tempo, tempo2));

                 if(tempo<tempo2){// verificando qual foi mais rápido na ordenação.

                       printf("\n(O Quicksort foi mais rapido na ordenacao).\n");

                        }

                          if(tempo2<tempo){

                               printf("\n(O MergeSort foi mais rapido na ordenacao).\n");

                          }else printf("\n[Quicksort]<time>(=)[MergeSort]\n");


            system("pause");

            sleep(1);// pausa de tempo.

            printf("\n\t\t\t*****VETOR MERGE*****\n\n");

            imprime(vetorMerge,size_vet);// imprime o vetor do MergeSort

            sleep(3);

            printf("\n\t\t\t*****VETOR QUICK*****\n\n");

            imprime(vetorQuick,size_vet);// imprimindo o vetor do Quicksort.

       return 0;

}
```

Para finalizar, essa última função chama a sequência de funções que serão executadas, depois uso as variáveis de tempo usadas no Quicksort e no MergeSort para apresentar na tela o tempo de execução dos algoritmos e a diferença de tempo entre os dois. Uso novamente a função de

busca para complementar as informações de tempo, e para que possa ser conferido posteriormente o resultado da busca nos vetores correspondentes. Após calcular e apresentar qual algoritmo foi mais rápido na ordenação será impresso os dois vetores ordenados para que seja possível conferir o resultado de busca.

Conclusão.

O Quicksort é considerado o algoritmo de ordenação mais eficiente que existe se tratando de uma quantidade não tão grande de elementos para serem ordenados. Aplicando esse algoritmo a uma linha independente de execução temos um resultado ainda melhor, já que cada recursão será executada de forma independente, pois não há necessidade de esperar uma chamada recursiva ser encerrada para que a outra execute. Temos então um ganho na eficiência, pois na teoria teríamos um lado do vetor sendo ordenado pela função, e o outro lado sendo ordenado pela mesma função, só que de forma recursiva.

No caso do MergeSort, além da divisão do vetor em partes, cada parte é subdividida em cada recursão, e em seguida a troca é feita. Esse algoritmo se destaca em grandes quantidades de números, pois gasta muita memória ao criar várias cópias da mesma função para as várias partes subdivididas do vetor, mas com o uso de linha de processamento independente cada função será executada em paralela com as demais funções criadas recursivamente. Esse pode ser o motivo no qual o MergeSort apresentou mais eficiência na ordenação.

Imagine um vetor de N elementos, esse vetor é dividido em duas partes e cada parte é ordenada ao mesmo tempo que a outra.

Agora imagine esse mesmo vetor dividido em duas partes, cada uma destas partes é subdividida em outras duas partes, cada uma destas quatro partes é ordenada ao mesmo tempo que as outras.

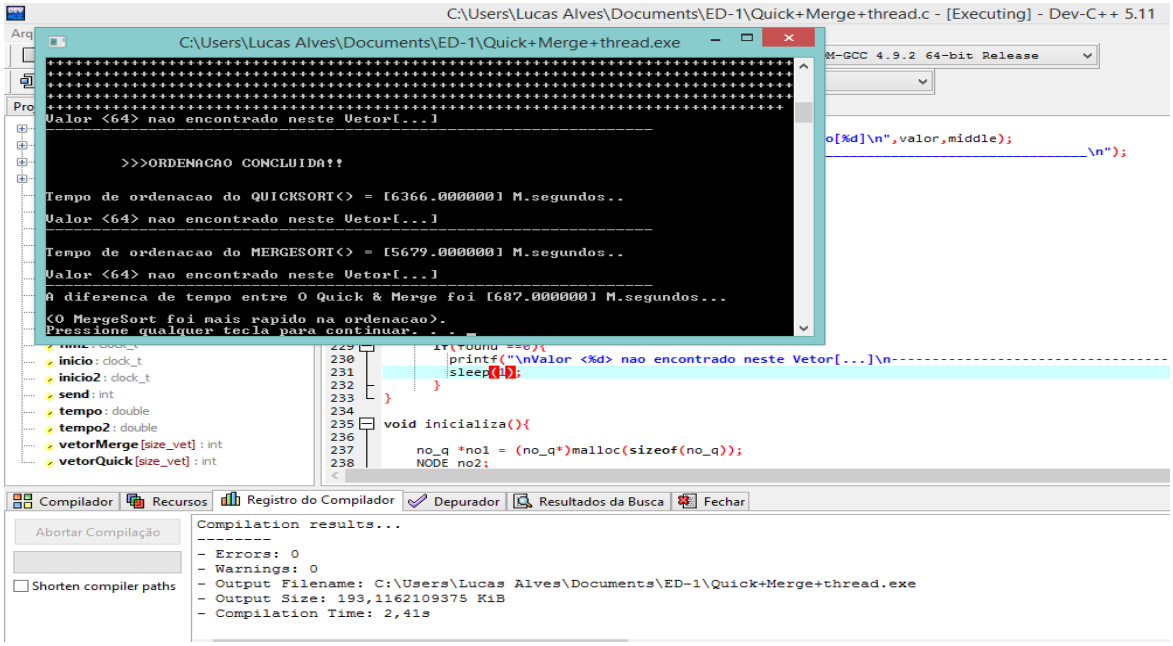
Também é importante ressaltar que a complexidade do MergeSort será sempre a mesma, tanto no melhor caso quanto no pior caso. Já o Quicksort pode sofrer alterações se tratando de sua complexidade.

Com isso dá para se ter uma ideia do porque o MergeSort se saiu melhor neste caso, esse algoritmo foi testado N vezes no computador com as seguintes configurações;

- Processador Celeron® CPU N2830 @ 2.16 GHz – 2.16 GHz;
- 4 GB de memória RAM;
- Sistema Operacional Microsoft Windows 8.1 Single Linguage em base de 64 bits-processador x64.

- Linguagem C.
- Software: DEV-C++ versão 5.11.

Na maioria dos testes o MergeSort apresentou o menor tempo, variando entre 5363.000000 e 8432.000000 microssegundos e o Quicksort apresentou na maioria dos testes o tempo maior, variando entre 6366.000000 e 102342.000000 microssegundos.
Resultado final:



Importante dizer que dependendo do tamanho do vetor e da capacidade do computador, esse resultado pode mudar dando vantagem ao QuickSort.

Referencias.

Fórum Script Brasil.
https://www.scriptbrasil.com.br/forum/topic/134859-uso-thread-com-c/

GitHub.
https://github.com/sangeeths/stackoverflow/blob/master/two-threads-parallel-merge-sort.c

Course Websites.
https://courses.engr.illinois.edu/cs241/fa2012/assignments/MergeSort/

Experts Exchange.
https://www.experts-exchange.com/questions/27937874/MergeSort-algorithm-using-Threads-in-JAVA.html

Programa com posix threads.
http://www.di.ubi.pt/~operativos/praticos/pdf/9-threads.pdf

Sistemas Operativos.
https://web.fe.up.pt/~pfs/aulas/so2012/at/5threads.pdf

Programação paralela e distribuída.
https://www.dcc.fc.up.pt/~ricroc/aulas/0708/ppd/apontamentos/pthreads.pdf

Clube do Hardware.
https://www.clubedohardware.com.br/forums/topic/1031279-resolvido-medir-tempo-de-execu%C3%A7%C3%A3o-em-c/

ACM@UIUC.
https://www-s.acm.illinois.edu/webmonkeys/book/c_guide/2.15.html