Relatório do Trabalho II de Programação Paralela

Diego Luiz N. Gonçalves[1],Thainara Orneles Matos[2], Ygor Takashi Nishi[3]

[1]Universidade Federal de Mato Grosso do Sul- Campus de Ponta Porã (UFMS-CPPP)
Res. Julia de Oliveira Cardinal – Ponta Porã– Mato Grosso do Sul – MS – Brazil

diegoreke@hotmail.com, orneles.thainara23@gmail.com,ygortn14@gmail.com

Resumo. *Este relatório tem como objetivo demonstrar a implementação e comparação de eficiência entre código sequêncial e os códigos em paralelo, usando a API CUDA, e a linguagem de programação C, na resolução do problema denominado "Parallel Dot Product".*
O problema consiste em 2 vetores de tamanho N, onde deve-se realizar a multiplicação de cada posição de um vetor com o outro, e depois realizar a soma do vetor resultante. Em todas as versões de códigos os vetores estão preenchidos com o valor 2.

Há 4 códigos no total:
1º é o código sequêncial.
2º Código em paralelo versão 2: Onde não usa-se a memória compartilhada, e está sendo usada a função atômica atomicAdd(), que é responsável por acrescentar(somar) a alguma varíavel, algum valor.
3º Código em paralelo versão 3: Onde usa-se a memória compartilhada, denominada por __shared__ e está sendo usada a função atômica $atomicAdd()$, que é responsável por acrescentar(somar) a alguma varíavel, algum valor.
4º Código em paralelo versão 4: Onde usa-se a memória compartilhada para realizar a multiplicação dos 2 vetores, e usa-se também a memória global para armazenar um vetor denominado 'd_c' que é responsável por guardar o resultado da operação de redução de cada bloco, ou seja, esse vetor terá um tamanho igual a quantidade de blocos, pois necessita armazenar o resultado da operação de redução de cada bloco.
Nesta versão em particular há 1 função do tipo $\_\_device\_\_$, que será utilizada pelo vetor d_c, e também há 2 chamadas de kernel para que todas as operações necessárias sejam realizadas.

1. Versão 1 - Código Sequêncial

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define TAM 8 //varíavel responsável por representar o tamanho do vetor

int main(){
    /*
      Sobre as variáveis:
      ->Estão do tipo "unsigned long int", sendo assim suportam apenas
```

```
    valores positivos, de 0 à 4.294.967.295
*/

unsigned long int *vetA, *vetB, i,resultado=0;

/*
  Sobre os vetores:
  ->Estão sendo alocados dinamicamente pela função malloc()
  ->>Motivo: Se tentar no modo tradicional possivelmente não
  conseguirá criar um vetor com posições acima de 1000000
*/

vetA = (unsigned long int*) malloc(TAM*sizeof(unsigned long int));
vetB = (unsigned long int*) malloc(TAM*sizeof(unsigned long int));

//Preenchendo os dois vetores com valor 2
for(i=0;i<TAM;i++){
    vetA[i] = vetB[i] = 2;
}

//--------------------------------- Inicia-se a contagem do tempo
clock_t begin = clock();

for(i=0;i<TAM;i++){
    /*
        Multiplicando vetor A & B na posição i, e armazenando o
        resultado no vetor A na posição i
    */
    vetA[i] = vetA[i]*vetB[i];

    /*
        Acrescentando o resultado da multiplicação na variável
        resultado
    */
    resultado+=vetA[i];
}

clock_t end = clock();
//--------------------------------- Encerra-se a contagem do tempo

/*
  OBS.: Para imprimir o conteúdo das variáveis do tipo unsigned
  long int usa-se %lu
*/
printf("Resultado: %lu\n",resultado);
printf("Tempo em (s):%f\n",(double)(end - begin) / CLOCKS_PER_SEC);
```

```
    return 0;
}
```

2. **Código Paralelo versão 2 - Utilizando a memória global, função atômica e apenas uma thread realizando a soma de todas as posições do vetor resultante**

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#define TAM 16777216 //aqui e o tamanho do vetor
#define THREADS_PER_BLOCK 1024 //threads por cada bloco

__global__ void dot(int *d_a, int *d_b, int *res_d){
    /*
        Note que aqui não há um vetor na memória compartilhada
    */

    int index = blockIdx.x * blockDim.x + threadIdx.x;

    d_a[index] = d_a[index] * d_b[index];

    __syncthreads(); //função que sincroniza as threads

    if(threadIdx.x == 0){
        int sum = 0;
        for(int i=0;i<THREADS_PER_BLOCK;i++){
            sum+=d_a[i];
        }
        atomicAdd(res_d,sum); //usando função atômica
    }

}

int main(){

    int *h_a, *h_b, *d_a, *d_b, *res_h, *res_d, i;

    h_a = (int*) malloc(TAM*sizeof(int));
    h_b = (int*) malloc(TAM*sizeof(int));
    res_h = (int*) malloc(sizeof(int));

    cudaMalloc((void**)&d_a,TAM*sizeof(int));
    cudaMalloc((void**)&d_b,TAM*sizeof(int));
```

```
    cudaMalloc((void**)&res_d,sizeof(int));

    for(i=0;i<TAM;i++){
        h_a[i] = h_b[i] = 2;
    }

    cudaMemcpy(d_a,h_a,TAM*sizeof(int),cudaMemcpyHostToDevice);
    cudaMemcpy(d_b,h_b,TAM*sizeof(int),cudaMemcpyHostToDevice);
    cudaMemcpy(res_d,res_h,sizeof(int),cudaMemcpyHostToDevice);


    //-------INICIA A CONTAGEM DO TEMPO

    clock_t begin = clock();

    dot<<<TAM/THREADS_PER_BLOCK,THREADS_PER_BLOCK>>>(d_a,d_b,res_d);

    clock_t end =  clock();

    //------ FINALIZA A CONTAGEM DE TEMPO

    cudaMemcpy(res_h,res_d,sizeof(int),cudaMemcpyDeviceToHost);

    printf("Resultado: %i\n",*res_h);
    printf("Tempo em(s):%f\n",(double) (end-begin)/CLOCKS_PER_SEC );

    free(h_a); free(h_b);
    cudaFree(d_a); cudaFree(d_b);

    return 0;
}
```

3. Código Paralelo versão 3 - Utilizando a memória compartilhada, função atômica e apenas uma thread realizando a soma de todas as posições do vetor resultante

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#define tam 16777216 //aqui e o tamanho do vetor
#define THREADS_PER_BLOCK 1024 //threads por cada bloco

__global__ void dot(int *d_a, int *d_b, int *res_d){
    /*
    Vetor de tamanho THREADS_PER_BLOCK, na memória compartilhada
```

```
    de cada bloco
    */
    __shared__ int d_c[THREADS_PER_BLOCK];

    int id= blockIdx.x * blockDim.x + threadIdx.x;
    d_c[threadIdx.x] = d_a[id] * d_b[id];

    __syncthreads(); //usando sincronização de threads

    if(threadIdx.x == 0){ //só entra aqui se for a 1ª thread
      int sum=0;
      for(int i=0;i<THREADS_PER_BLOCK;i++){
          sum+=d_c[i];
      }
      atomicAdd(res_d,sum); //usando a função atômica
    }

}


int main(){

    /*------- Variáveis:
      h_a: 1º vetor de inteiros da CPU
      h_b: 2º vetor de inteiros da CPU
      d_a: 1º vetor de inteiros da GPU
      d_b: 2º vetor de inteiros da GPU
      res_h: varíavel para guardar o resultado na CPU
      res_b: varíavel para guardar o resultado na GPU
    */

    int *h_a, *h_b, *d_a, *d_b, i, *res_h,*res_d;
    /*
        Alocando espaço na memória da CPU
        para os vetores h_a, h_b e para a
        variável res_h
    */
    h_a = (int*) malloc(tam*sizeof(int));
    h_b = (int*) malloc(tam*sizeof(int));
    res_h = (int*) malloc(sizeof(int));

    /*
        Alocando espaço na memória da GPU
        para os vetores d_a, d_b e varíavel
        res_d
    */
```

```
    cudaMalloc((void**)&d_a,tam*sizeof(int));
    cudaMalloc((void**)&d_b,tam*sizeof(int));
    cudaMalloc((void**)&res_d,sizeof(int));

    //Iniciando os vetores do host com valor 2
    for(i=0;i<tam;i++){
        h_a[i] = h_b[i] = 2;
    }


    cudaMemcpy(d_a,h_a,tam*sizeof(int),cudaMemcpyHostToDevice);
    cudaMemcpy(d_b,h_b,tam*sizeof(int),cudaMemcpyHostToDevice);
    cudaMemcpy(res_d,res_h,sizeof(int),cudaMemcpyHostToDevice);


    //--------Inicío contagem de tempo

    clock_t begin = clock();

    dot<<<tam/THREADS_PER_BLOCK,THREADS_PER_BLOCK>>>(d_a,d_b,res_d);

    clock_t end = clock();

    //--------FIM contagem de tempo

    cudaMemcpy(res_h,res_d,sizeof(int),cudaMemcpyDeviceToHost);

    printf("Resultado: %i\n",*res_h);
    printf("Tempo em (s):%f\n",(double) (end-begin)/CLOCKS_PER_SEC);
    /*
        Liberando a memória da CPU (função free()) e GPU(cudaFree())
    */
    free(h_a); free(h_b);
    cudaFree(d_a); cudaFree(d_b);

    return 0;
}
```

4. **Código Paralelo Versão 4 - Utilizando memória compartilhada e global, 1 função de device, 2 funções de kernel, 2 chamadas de kernel e um vetor para armazenar o resultado do vetor na memória compartilhada de cada bloco**

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

```
#include <time.h>

const int TAM = 1048576; //Tamanho do vetor
const int THREADS_PER_BLOCK = 1024; // Threads em cada bloco
const int BLOCKS = TAM/THREADS_PER_BLOCK; //Nº de blocos

/*
    -> Função de __device__ que somente será usada pela GPU
    --> Esta função tem como objetivo preencher o vetor d_c
    com os resultados de cada bloco (resultante da operação
    de redução de cada bloco).

    ->> Parâmetros:
        vet: varíavel inteira que irá receber o valor da
        posição 0 do vetor compartilhado dentro do bloco
        que chamou esta função

        d_c: o vetor onde será inserido o valor (vet)

        position: para saber em qual posição de d_c, vet
        será ser inserido
*/
__device__ void put_operation(int vet, int *d_c, int position){
    // Inserindo o valor vet no vetor d_c na posição 'position'
    d_c[position] = vet;
}

/*
    -> Função de kernel que será chamada pela CPU para realizar
    a multiplicação dos vetores d_a e d_b e armazenar o resultado
    dentro de um vetor na memória compartilhada __shared__ dentro
    de cada bloco, sendo que neste vetor será realizada a operação
    de redução, e após isso será chamada a função __device__
    put_operation() para que o vetor d_c seja preeenchido de acordo
    com os resultados da operação de redução de cada bloco.

    -> Parâmetros:
       d_a: Endereço do 1º vetor que está alocado na memória global
       (GPU)

       d_b: Endereço do 2º vetor que está alocado na memória global
       (GPU)

       d_c: Endereço do 3º vetor que está alocado na memória global
       (GPU)
*/
```

```
__global__ void dot(int *d_a, int *d_b, int *d_c){
    /*Varíavel index é responsável por representar os índices dos
    vetores d_a & d_b (que estão na memória global)*/
    int index = threadIdx.x;

    /*
        Vetor na memória compartilhada de cada bloco responsável por
        armazenar o resultado da multiplicação dos vetores d_a & d_b
    */
    __shared__ int vet[THREADS_PER_BLOCK];

    //Realizando a operação de multiplicação dos vetores
    vet[index] = d_a[index] * d_b[index];

    /*
        Sincronizando as threads dos blocos para garantir que todas
        as posições do vetor vet estejam preenchidas
    */
    __syncthreads();


    int salto = 1; //Tamanho do salto a cada iteração
    /*
        Variável de controle para limitar a quantidade de threads
        que estarão ativas a cada iteração
    */
    int nmr_threads = blockDim.x/2;

    /*
        Este laço será executado enquanto o nº de threads (variável nmr_threads) for
    */
    while(nmr_threads > 0){
        /*
            Se o índice da thread for menor que o valor da variável
            de controle nmr_threads então executa.
            Este if garante que apenas as threads que devem estar
            ativas, vão estar
        */
        if(threadIdx.x < nmr_threads){
            /*
                -> Variaveis:
                first: variável que representa o índice do vetor
                compartilhado no bloco, onde será armazenado o
                resultado

                second: variável que representa o índice do vetor
```

```
                compartilhado no bloco, que é a posição de onde o
                valor a ser somado será extraído
            */
            int first = index * salto *2;
            int second = first + salto;
            vet[first] += vet[second];
        }
        //A cada iteração o salto dobra
        salto = salto * 2;

        //A cada iteração o nmr_threads diminui pela metade
        nmr_threads =  nmr_threads / 2;

        /*
            Sincronizando as threads para evitar erros de posições
            não preenchidas
        */
        __syncthreads();
    }
    /*
        Somente a thread 0 deverá invocar a função de __device__
        put_operation(), para que o vetor d_c seja preenchido com o
        resultado da redução que está armazenado em vet[0]
    */
    if(index == 0)
      put_operation(vet[0], d_c, blockIdx.x);
}

/*
    -> Função de kernel que realiza a mesma redução realizada na
    função acima, e neste caso está sendo utilizada somente pelo
    vetor d_c
*/
__global__ void reduction_operation(int *vet){

    int index = threadIdx.x;
    int salto = 1;
    int nmr_threads = blockDim.x/2;

    while(nmr_threads > 0){
        if(index < nmr_threads){
            int first = index * salto *2;
            int second = first + salto;
            vet[first] += vet[second];
        }
```

```
        salto = salto * 2;
        nmr_threads =  nmr_threads / 2;
        __syncthreads();
    }

}

int main(){
   /*
        ====>> Variáveis
        *h_a : ponteiro do tipo inteiro, para guardar o 1º vetor
        de inteiros na CPU

        *h_b : ponteiro do tipo inteiro, para guardar o 2º vetor
        de inteiros na CPU

        *d_a : ponteiro do tipo inteiro, para guardar o 1º vetor
        de inteiros na GPU

        *d_b : ponteiro do tipo inteiro, para guardar o 2º vetor
        de inteiros na GPU

        *d_c : ponteiro do tipo inteiro, para guardar o 3º vetor
        de inteiros na GPU, este vetor será utilizado para guardar
        os resultados finais de cada bloco, portanto terá o mesmo
        tamanho da quantidade de blocos declarada no programa

   */
    int *h_a, *h_b ,*d_a, *d_b, *d_c, i;

    /*
        Alocando espaço na memória da CPU
        para os vetores h_a, h_b
    */
    h_a =(int*) malloc(TAM*sizeof(int));
    h_b =(int*) malloc(TAM*sizeof(int));

    /*
        Alocando espaço na memória da GPU
        para os vetores d_a, d_b, d_c
    */
    cudaMalloc((void**) &d_a, TAM*sizeof(int));
    cudaMalloc((void**) &d_b, TAM*sizeof(int));
    cudaMalloc((void**) &d_c, BLOCKS*sizeof(int));

    //Iniciando os vetores do host com valor 2
```

```
    for(i=0;i<TAM;i++){
        h_a[i] = h_b[i] = 2;
    }

    /*
        Realizando a cópia dos valores dos vetores
        h_a e h_b do host (CPU), para o device (GPU)
    */
    cudaMemcpy(d_a,h_a,TAM*sizeof(int),cudaMemcpyHostToDevice);
    cudaMemcpy(d_b,h_b,TAM*sizeof(int),cudaMemcpyHostToDevice);

    /*====================INÍCIO CONTAGEM DE TEMPO=======*/
    clock_t begin = clock();

    dot<<<BLOCKS,THREADS_PER_BLOCK>>>(d_a, d_b, d_c);
    /*
        Após realizar todas as operações com os vetores
        d_a & d_b, o resultado de cada bloco estará
        armazenado no vetor d_c, portanto também será
        necessário realizar a operação de redução com
        o mesmo, então foi criado outra função de kernel
        que realiza apenas a operação de redução.
        OBS: Foi necessário criar esta função como uma
        função de kernel pois se fosse criado como uma
        de device (para reutilizar código) não teria como
        realizar a redução do vetor d_c (esta forma não foi
        encontrada), sendo assim, infelizmente, foi necessário
        repetição de código, pois não temos conhecimento até o
        momento de como sincronizar os blocos.

    */
    reduction_operation<<<1,THREADS_PER_BLOCK>>>(d_c);

    clock_t end = clock();
    /*==================FIM CONTAGEM DE TEMPO=======*/

    // Copiando os valores do vetor d_c para h_a
    cudaMemcpy(h_a,d_c,BLOCKS*sizeof(int),cudaMemcpyDeviceToHost);

    printf("Resultado: %i\n",h_a[0]);
    printf("Tempo em (s): %f\n",(double) (end-begin)/CLOCKS_PER_SEC );

    /*
        Liberando a memória da CPU (função free()) e GPU(cudaFree())
    */
    free(h_a); free(h_b);
```

```
    cudaFree(d_a); cudaFree(d_b);

    return 0;
}
```

5. Comparativos de tempo

5.1. Tabela de tempo: Código sequêncial e códigos paralelos

Entrada	CS ($segundos$)	CPv2 ($segundos$)	CPv3 ($segundos$)	CPv4 ($segundos$)
2^{10}	0.000005	0.00008425	0.0000785	0.00008
2^{11}	0.000009	0.00008225	0.000083	0.000082
2^{12}	0.000017	0.0000775	0.0000805	0.000081
2^{13}	0.00003325	0.000074	0.00007475	0.000095
2^{14}	0.0000648	0.0000815	0.000075	0.000097
2^{15}	0.000129	0.00008175	0.0000805	0.000091
2^{16}	0.000261	0.00007825	0.0000825	0.000119
2^{17}	0.0005265	0.00008475	0.00008	0.000143
2^{18}	0.0010645	0.00008225	0.00008625	0.000222
2^{19}	0.00213425	0.00009125	0.00009	0.000260
2^{20}	0.00438425	0.00009175	0.0000875	0.000254

Tabela 1. Legenda: CS = Código Sequêncial, CPv2 = Código Paralelo versão 2, CPv3 = Código Paralelo versão 3

5.2. Tabela de Speed up

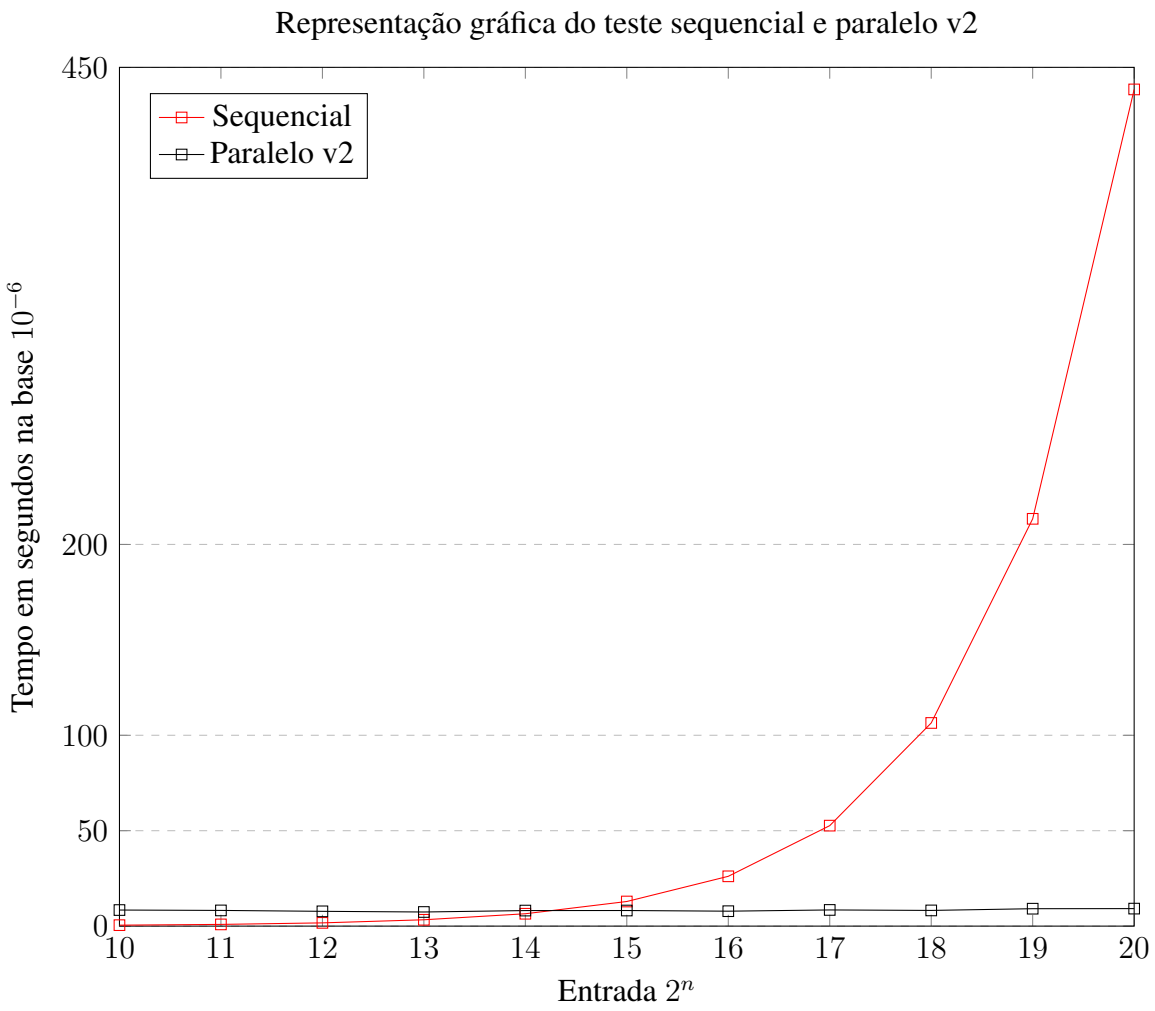
Entrada	Speed up *CS vs. CPv2*	Speed up *CS vs. CPv3*	Speed up *CPv2 vs. CPv3*
2^{10}	0.059	0.063	1.073
2^{11}	0.109	0.108	0.990
2^{12}	0.219	0.2	0.911
2^{13}	0.449	0.444	0.989
2^{14}	0.795	0.864	1.086
2^{15}	1.577	1.602	1.015
2^{16}	3.335	3.163	0.948
2^{17}	6.212	6.581	1.059
2^{18}	12.942	12.342	0.953
2^{19}	23.389	23.713	1.013
2^{20}	47.784	50.105	1.048

Tabela 2. Legenda: CS = Código Sequêncial, CPv2 = Código Paralelo versão 2, CPv3 = Código Paralelo versão 3

Entrada	Speed up *CS vs. CPv4*	Speed up *CPv2 vs. CPv4*	Speed up *CPv3 vs. CPv4*
2^{10}	0.062	1.053	0.981
2^{11}	0.109	1.003	1.012
2^{12}	0,209	0.956	0,993
2^{13}	0.35	0.778	0,786
2^{14}	0.668	0.840	0,773
2^{15}	1.417	0.898	0.884
2^{16}	2.193	0.657	0.693
2^{17}	3.681	0.592	0.559
2^{18}	4.795	0,370	0.388
2^{19}	8.208	0,350	0.346
2^{20}	17.260	0.361	0.344

Tabela 3. Legenda: CS = Código Sequêncial, CPv2 = Código Paralelo versão 2, CPv3 = Código Paralelo versão 3, CPv4 = Código Paralelo versão 4

6. Gráficos comparativos



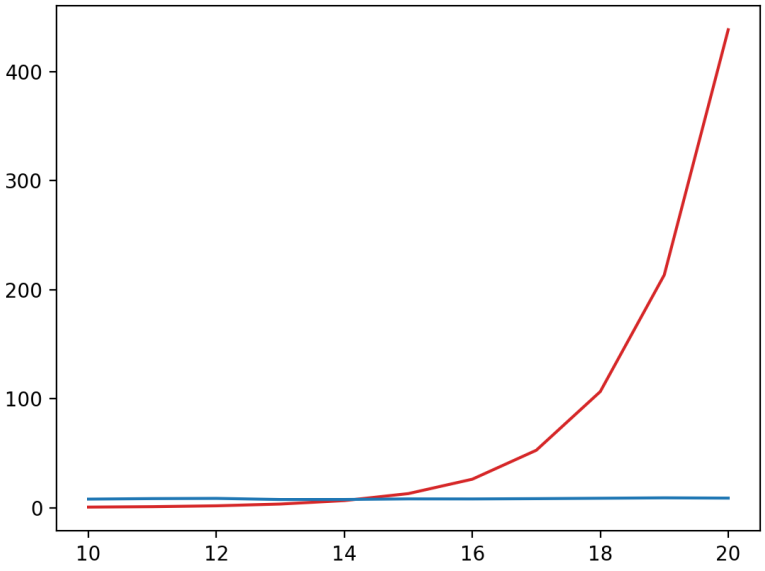


Figura 1. Código Sequêncial(Vermelho) vs. Paralelo Versão 3(Azul). Eixo x = Entradas 2^n onde $n =$ número do eixo x. Eixo y = Tempo em segundos

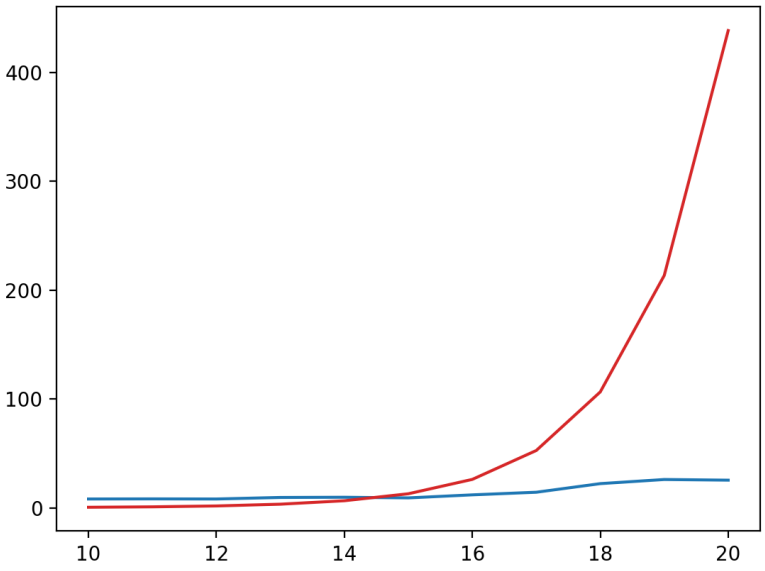


Figura 2. Código Sequêncial(Vermelho) vs. Paralelo Versão 4(Azul). Eixo x = Entradas 2^n onde $n =$ número do eixo x. Eixo y = Tempo em segundos