CONGRESSO CATARINENSE DE CIÊNCIA DA COMPUTAÇÃO
2019. Ano 6, Vol. 6. ISSN: 2319-0655

# Counting Sort paralelizado com OpenMP

**Felipe de Lima Peressim[1], Rodrigo Curvello[1]**

[1]Acadêmico Bacharel Ciência da Computação – Instituto Federal de Educação, Ciência e Tecnologia Catarinense (IFC) – Câmpus Rio do Sul
Cep 89160-202 – Rio do Sul – SC – Brasil

fe.peressim@gmail.com, rodrigo.curvello@ifc.edu.br

***Abstract.*** *This work aims to investigate the efficiency that can be obtained by high performance computing. In this sense, the integer sorting algorithm Counting Sort is used as research object. Implemented in the C language and parallelized through the API OpenMP. The efficiency comparisons of are performed through the speedup calculation obtained by the execution times of the serial and parallel implementations of the so algorithm.*

***Key-words:*** *Parallelism; Concurrency; Thread; Counting Sort.*

***Resumo.*** *Este trabalho propõe-se investigar a eficiência que pode se obter através da programação de alto desempenho. Neste sentido, o algoritmo de ordenação de inteiros Counting Sort é utilizado como objeto de pesquisa. Sendo implementado na linguagem de programação C e paralelizado através da API OpenMP. As comparações de eficiência são realizadas através do cálculo de speed-up obtido através dos tempos de execução das implementações serial e paralela do algoritmo em questão.*

***Palavras-chave:*** *Paralelismo; Concorrência; Thread; Counting Sort.*

## 1. Introdução

A medida que o número de núcleos aumenta e os transistores diminuem a lei de Moore vai se cumprindo. Por consequência deste aumento uma porcentagem maior da CPU acaba não sendo utilizada por algoritmos de fluxo sequencial (DUVANENKO, 2010). Desta forma é necessário prover-se de ferramentas que permitam executar algoritmos de forma concorrente / paralela para tirar proveito máximo dos processadores no que tange o uso e o desempenho que se pode alcançar através do multiprocessamento.

A concorrência computacional permite uma vantagem na velocidade de execução sobre computação sequencial. Programas sequencias realizam apenas uma tarefa por vez, isso é, a tarefa é proporcional ao total de operações realizadas. No entanto é possível decompor o programa em pedaços relativamente independentes que precisem se comunicar apenas raramente, inclusive é possível separar as tarefas para executarem em processadores diferentes, produzindo vantagens de velocidades proporcionais ao número de processadores (ABELSON e SUSSMAN, 1996, p. 298).

Alto desempenho e Eficiência são vantagens oriundas da concorrência. De acordo com Navaux

(2001) o alto desempenho pode ser obtido quando se faz o uso do processador em sua totalidade, ou seja, enquanto um processo realiza alguma operação de entrada e saída - geralmente de natureza bloqueante - algum outro processo pode utilizar o processador para realizar alguma outra tarefa, portanto acaba por reduzir o tempo total de execução. Em relação a eficiência Navaux (2001) menciona o uso ótimo de uma determina arquitetura de processamento, i.e utilizar-se ao máximo de todos os recursos providos por tal arquitetura na realização do processamento para atingir o alto desempenho e aumentar o número de tarefas realizadas em um intervalo de tempo.

Com base nas afirmações supracitadas, o presente trabalho propõe-se realizar um estudo de caso para verificar o desempenho e a vantagem computacional que pode ser obtida através da programação paralela. Neste estudo duas versões do algoritmo de ordenação *Counting Sort* são implementadas, uma utilizando a programação sequencial e a outra a programação paralela através da *API OpenMP*. E a verificação do desempenho é realizada através do calculo de *speed-up* dos algoritmos em questão.

A escolha do algoritmo se deu por dois motivos principais: O primeiro é a importância dos algoritmos de ordenação que são fundamentais em muitas áreas da Ciência da Computação devido a sua extensa aplicação; o segundo é devido a possibilidade de paralelizar o algoritmo que por sua vez tornar-se objeto de estudo ideal para os propósitos deste trabalho, sendo tal possibilidade constatada por evidências observadas na literatura (DUVANENKO, 2010), (KAUL; VIDYAPEETH'S, 2015) e (FAUJDAR; GHRERA, 2016).

Em relação a escolha da *API OpenMP* dá-se devido a está especificação prover um modelo de programação paralela que é portável entre arquiteturas diferentes, além do suporte que possui por parte de diferentes compiladores. Além disso ao se utilizar esta ferramenta, os algoritmos tornam-se menos tendenciosos a erros porque a responsabilidade de realizar o paralelismo é atribuída à *API,* e portanto o programador apenas se preocupa em indicar regiões do código que devem ser paralelizadas.

## 2. Fundamentação Teórica

### 2.1. Arquiteturas paralelas e Paralelismo

Uma das primeiras classificações de arquitetura paralela que pode ser utilizada de forma geral é a classificação de Flynn (1972). Nesta classificação conforme Flynn (1972) toma-se como base a execução de uma sequência de instruções sobre uma sequência de dados em um computador. Neste modelo diferencia-se o fluxo de instruções e o fluxo de dados. Através de uma permutação das possibilidades de sequencia de instruções e de dados Flynn (1972) propôs quatro classes de arquiteturas paralelas: SISD, SIMD, MISD, MIMD.

Conforme Navaux (2001) na classe SISD (*Single Instruction Single Data*) "um único fluxo de instruções atua sobre um único fluxo de dados.".

De acordo com Rose e Navaux (2001) na arquitetura de máquinas SIMD (*Single Instruction Multiple Data*) "[...] uma única instrução, através de uma unidade de controle, atua de forma síncrona sobre um conjunto de dados diferentes, distribuídos ao longo de processadores elementares.

Navaux (2001) também explica que na classe MISD (*Multiple Instruction Single Data)* "múltiplos fluxos de instruções atuariam sobre um único fluxo de dados". De acordo com o mesmo,

"[..] em uma máquina MIMD (*Multiple Instruction Multiple Data*), cada unidade de controle C recebe um fluxo de instruções próprio e repassa-o para sua unidade processadora P, para que seja executado sobre um fluxo de instrução próprio".

Uma outra forma de categorizar a classificação das máquinas paralelas é de acordo com o compartilhamento de memória. De acordo com Silberschatz, Galvin e Gagne (2008), na memória compartilhada existe um único espaço de endereçamento que é utilizado de forma implícita na comunicação entre os processadores envolvidos no processo. No caso de memória não compartilhada Rose e Navaux (2001) afirmam que existem múltiplos espaços de endereçamento privados, um para cada processador.

Além destas, os autores também comentam sobre a memória distribuída e a memória centralizada, que neste caso diz respeito a proximidade da memória do processador. Além disso uma característica essencial da comunicação dá-se na relação de acesso às memórias de máquinas multiprocessadas quais comentadas conforme Rose e Navaux (2001): acesso uniforme à memória (*uniform memory access,* UMA) e acesso não uniforme à memória (*non-uniform memory access*, NUMA) e NORMA (*non- remote memory access*) acesso não remoto à memória, nesta última o acesso é feito através de mensagens.

Conforme Rose e Navaux (2001) no modelo UMA a memória é centralizada e encontra-se a mesma distância de todos os processadores, ou seja, desta forma o acessa a memória tem a mesma latência para todos os processadores. Já no modelo NUMA a memória é distribuída, de modo que sua implementação permite múltiplos módulos que são associados um a cada processador.

A Tabela 1 exibe as arquiteturas comentadas, o tipo de comunicação destas e o tipo de acesso das mesmas.

| **Tipo** | **SIMD** | **PVP** | **SMP** | **MPP** | **DSM** | **Cluster** |
|---|---|---|---|---|---|---|
| Estrutura | SIMD | MIMD | MIMD | MIMD | MIMD | MIMD |
| Comunicação | direta | memória compartilhada | memória compartilhada | troca de mensagem | memória compartilhada | troca de mensagem |
| Acesso à memória | UMA | UMA | UMA | NORMA | NUMA | NORMA |

**Tabela 1: Modelos de Arquiteturas Paralelas: Fonte: Adaptado dê Rose e Navaux (2001)**

## 2.2. Tipos de paralelismo

As arquiteturas explicitadas neste presente trabalho trabalham em conformidade com os modelos de programação paralela designados para tal. De acordo com Navaux (2001) estes modelos se dividem em um modelo de programação implícita e três modelos de programação explícita: Passagem de Mensagens, Paralelismo de Dados e Variáveis Compartilhadas.

Conforme Navaux (2001) conceitua, o Paralelismo Implícito ocorre quando o compilador ou o sistema são responsáveis por determinar quais partes do programa devem ser paralelizadas. Já no Paralelismo de dados, Navaux (2001) afirma que estes modelo pode ser usado nos modos de execução SIMP ou SPMD. Neste modelo a mesma instrução ou parte do programa é executada em

nodos de processamento diferentes.

No modelo de Passagem de Mensagens de acordo com Silberschatz, Galvin e Gagne (2008) diferentes processos cada qual com seu contexto devem estar em sincronia um para com o outro. E conforme Navaux (2001) este modelo pode suportar tanto paralelismo de controle (MPMD) como de dado (SPMD). E também conforme o mesmo autor no modelo de Variáveis Compartilhadas existe um espaço de endereçamento único onde a comunicação é feita através da escrita e leitura de variáveis compartilhadas e a sincronização é explícita.

De acordo com Sena e Costa (2008) o processamento paralelo pode ser realizado em ambientes de memória distribuída e ambientes de memória compartilhada.

O ambiente de memória distribuída de acordo com os autores consiste de vários processadores que possuem seus próprios recursos de memória e são interconectados por uma rede de comunicação e o processamento ocorre através da troca de dados. Geralmente essa comunicação é feita através da troca de mensagens entre as tarefas iniciadas e distribuídas pelos processadores do ambiente, cada uma utilizando seu próprio endereçamento de memória.

Em contrapartida no ambiente de memória compartilhada os processadores compartilham o espaço de endereçamento de uma única memória. Embora neste ambiente os processadores compartilhem o mesmo endereçamento a complexidade é menor, porque mesmo embora estes possam operar de forma independente, porém todas as mudanças realizadas no espaço de memória tornam-se visíveis a todos os processadores.

### 2.3. OpenMP

OpenMP é uma *API* que suporta programação paralela de memória compartilhada multiplataforma com suporte as linguagens de programação C/C++ e Fortran. Está API define um modelo portável e escalável através de uma interface simples e flexível para o desenvolvimento de aplicações paralelas em plataformas que variam de computadores pessoais até supercomputadores (OPENMP, 2019).

De acordo com Sena e Costa (2008) OpenMP é um padrão desenvolvido e mantido pelo grupo OpenMP *Architecture Review Board* (ARB), constituído pelos grandes fabricantes de software e hardware do mundo, tais como SUN Microsystems, SGI, IM, Intel, dentre outros. Em 1997 estes fabricantes se reuniram e criaram um padrão de programação paralela para arquiteturas de memória compartilhada.

O OpenMP é constituído de uma *API* e um conjunto de diretivas que permite a confecção de algoritmos paralelos com compartilhamento de memória através da implementação automática e otimizada de um conjunto de *Threads*. Para isso o padrão define como os compiladores devem gerar os códigos de execução paralela através da incorporação, nos programas sequências, de diretivas que indicam como as tarefas deverão ser divididas entre os processadores ou núcleos (SENA; COSTA, 2008).

Existem dois benefícios no uso da abordagem de diretivas: A primeira é que esta abordagem permite que o código base seja utilizado em ambas plataformas de mono processamento e multiprocessamento; na primeira as diretivas são simplesmente tratadas como comentários e ignoradas pelo tradutor da linguagem, permitindo uma execução correta da versão serial; O segundo benefício se dá devido a abordagem incremental ao paralelismo que a *API* permite -

iniciando de um programa sequencial, o programador especifica através das diretivas as partes do código em que o paralelismo deverá ser expresso (CHANDRA, 2000).

Além das vantagens supracitadas o uso do OpenMP oferece diversas outras vantagens, dentre as quais de acordo Sena e Costa (2008) destacam-se: poucas alterações no código serial existente; robusta estrutura para suporte a programação paralela; fácil compreensão e uso das diretivas; suporte a paralelismo aninhado e possibilidade de ajuste dinâmico do número de *Threads*.

## 2.4. Counting Sort

Em 2009, Cormen et al., introduziu um novo algoritmo em seu livro *Introduction to Algorithms.* O algoritmo em questão não utilizava um método de comparação para ordenar uma lista. Sua estratégia se baseia em contar a ocorrência dos valores da lista a priori e realizar a ordenação a posteriori. Portanto este algoritmo fora denominado de *Counting Sort* - Ordenação por contagem (SAIAN, 2018).

De acordo com Faujdar e Ghrera (2016) o algoritmo *Counting Sort* é um algoritmo para ordenação de números inteiros. Este algoritmo é simples e eficiente com complexidade assintótica de tempo linear $O(n + k)$ onde $n$ é a quantidade de elementos de entrada e $k$ é o intervalo de 0 a $k$ ao qual pertencem os $n$ elementos.

Conforme Cormen et al., (2001) quando $k = O(n)$ então o algoritmo é executado em tempo $O(n)$. Adicionalmente *Counting Sort* é um algoritmo estável, ou seja, se um mesmo elemento ocorrer mais do que uma vez nos dados, a ordem dos elementos duplicados é mantida.

Cormen et al., (2001) explica que a ideia básica do algoritmo é determinar, para cada elemento de entrada $x$, o número de elementos menores que $x$. Essa informação pode ser utilizada para colocar $x$ diretamente na posição correspondente no vetor de saída. Por exemplo, se existem 10 elementos menores que $x$, então $x$ pertence a posição 11. Este esquema deve ser ligeiramente modificado quando houver vários elementos com o mesmo valor. A listagem 1 exibe o pseudocódigo do algoritmo em questão.

```
0 for i ← 0 to k
1   do C[i]←0
2
3 for j ← 1 to length[A]
4   do C[A[j]] ← C[A[j]] + 1
5
6 for i ← 1 to k
7   do C[i] ← C[i] + C[i - 1]

9 for j ← length[A] downto 1
10    do B[C[A[j]]] ← A[j]
11      C[A[j]] ← C[A[j]] - 1
```

**Listagem 1: Pseudoalgoritmo do Counting Sort. Fonte: Adaptado de Cormen et al., (2001).**

Basicamente o algoritmo realiza 4 etapas para realizar a ordenação, sendo estas as seguintes:

1. Inicializa os elementos do vetor auxiliar com zeros.
2. Utiliza os valores do vetor de entrada como índice no vetor auxiliar para contar as ocorrências.
3. Ordena as posições não vazias do vetor auxiliar.
4. Transfere os valores do vetor auxiliar para o vetor de saída.

O pseudocódigo destacado na Listagem 1 utiliza três vetores no processo de ordenação. Um vetor *A* de entrada com *n* elementos inteiros. Como *A* possuí *n* elementos, logo um vetor *B* com *n* elementos faz-se necessário para armazenar os valores ordenados de *A*. Um terceiro vetor auxiliar *C* é necessário para armazenar as ocorrências dos *n* elementos de *A*. O vetor *C* é alocado com tamanho para alocar *k* elementos, onde *k* é o maior inteiro encontrado em *A*.

Inicialmente na linha 0 as posições de *C* são inicializadas com zeros - inicialmente todos os *k* elementos são considerados com ocorrência nula. Na linha 3 o vetor *A* é percorrido elemento a elemento, e na linha 4 os elementos de *A* são utilizados como índices para acessar as posições que estes elementos representam como índices do vetor *C* para armazenar as ocorrências dos elementos em *C*.

Na linha 6 o vetor C é percorrido elemento a elemento e na linha 7 o vetor auxiliar de frequências é modificado para armazenar em cada posição *i* a soma das frequências anteriores. O vetor é modificado para indicar a posição de cada elemento no vetor de saída. Basicamente para cada índice $i = 0,1,...,$ *k* é determinado quantos elementos aparecem antes de *i* através da soma das contagens do vetor *C*.

Finalmente da linha 9 em diante o vetor é ordenado. Cada elemento *A[j]* é armazenado em sua posição ordenada no vetor de saída *B*. Se todos os *n* elementos são distintos, então a partir da linha 9 para cada elemento *A[j]*, o valor *C[A[j]]* é a posição final correta de *A[j]* no vetor de saída *B*. Devido a possível ocorrência de elementos distintos, as frequências *C[A[j]]* são decrementadas cada vez que um valor de *A[j]* é armazenado no vetor *B*. Decrementando *C[A[j]]* faz com que o próximo elemento de entrada com um valor igual a *A[j]*, caso este exista, ir para a próxima posição imediatamente antes de *A[j]* no vetor de saída *B*.

Para fins de compreensão considere um vetor *A* arbitrário com 16 elementos a ser ordenado com os seguintes valores de entrada:

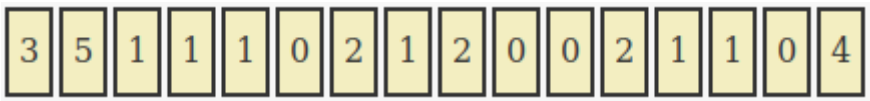


**Figura 1: Vetor *A* de inteiros não ordenado. Fonte: Elaborado pelo autor.**

Através da Figura 1 é possível observar que o maior valor do vetor *A* é o 5, portanto o vetor de frequências deverá ser um vetor alocado com 6 posições, i.e de 0 à 5 têm-se 6 possíveis valores.

Uma vez que as contagens foram realizadas tem-se o vetor resultante *C* ilustrado pela Figura 2:

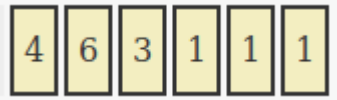


**Figura 2: Vetor *C* de frequências. Fonte: Elaborado pelo autor.**

Como se pode observar pela Figura 2 cada posição do vetor possuí as frequências dos valores

não ordenados, ou seja, o inteiro 0 ocorre quatro vezes dentro do vetor *A*, já o inteiro 1 ocorre seis vezes, o inteiro 2 ocorre três vezes, em contrapartida os inteiros 3, 4 e 5 ocorrem apenas uma vez.

O próximo passo do algoritmo uma vez que o vetor de frequências foi construído é somar valores adjacentes do vetor de frequências para sinalizar quantos números ocorrem antes de um determinado valor, desta forma cada inteiro pode ser ordenado em suas correspondentes posições.

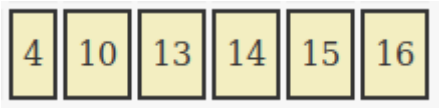


**Figura 3 Vetor *C* de frequências. Fonte: Elaborado pelo autor.**

Inicialmente o valor 6 contido no índice 1 do vetor *C* da Figura 2 que diz respeito as frequências do inteiro 1 é somado ao inteiro 4 contido no índice 0 do vetor e assim produzindo o inteiro 10, que é armazenado na posição 1 do vetor substituindo o antigo valor de ocorrência 6.

Esse processo pois no momento da ordenação o vetor original é percorrido novamente e portanto ao encontrar o inteiro 1 pela primeira vez no vetor saber-se-á que antes deste inteiro ocorrem outros dez valores contando com o mesmo, isso é os outros 4 zeros e os respectivos 1 que dever-se-ão aparecer antes dele. No processo descrito então, o inteiro 10, presente no índice 1 do vetor *C* é utilizado como índice para alocar o primeiro valor 1 no vetor de ordenação *B* que alocar-se-á os valores ordenados. Uma vez que o primeiro inteiro 1 é alocado na posição 10, o inteiro 10 é decrementado, ou seja, o próximo inteiro 1 do vetor não ordenado que for encontrado será alocado na posição 9 do vetor *B* e assim por diante.

O mesmo processo ocorre para todos os outros valores do vetor. Depois de os valores 4 e 6 serem somados produzindo o valor 10, o valor 10 é somado com o inteiro a sua direita, i.e 3, gerando se o inteiro 13, os demais valores tem ocorrência 1, portanto somando-se ao inteiro 13 vão respectivamente produzindo os inteiros 14, 15 e 16. Após todo o processo de contagem e ordenação ocorrer, ter-se-á o vetor ordenado *B* conforme ilustrado pela Figura 4.


**Figura 4 Vetor *B* resultante de *A* ordenado. Fonte: Elaborado pelo autor.**

## 2.5. Implementação

Nesta subseção algumas particularidades da implementação paralela são discutidas, no entanto apenas parcelas do algoritmo são apresentadas. O código completa se encontra no seguinte repositório hospedado no guithub: http://github.com/feperessim/counting-sort-parallel-openmp/

```
0 void couting_sort_parallel(unsigned int A[], unsigned int B[], unsigned int C[]) {
1 #pragma omp parallel num_threads(NUM_THREADS) shared(A, B, C)
2 {
```

```
3   #pragma omp for
4   for (int i = 0; i < MAX_INTEGERS_TO_SORT; i++) {
5     C[A[i]] = C[A[i]] + 1;
6   }
7   #pragma omp for
8   for (int i = 1; i < MAX_INTEGER_RANGE + 1; i++) {
9     C[i] = C[i] + C[i - 1];
10    }
11    #pragma omp for
12    for (int i = MAX_INTEGERS_TO_SORT - 1; i >= 0; i--) {
13      B[C[A[i]] -1] = A[i];
14      C[A[i]] =  C[A[i]] - 1;
15    }
16  }
17}
```

**Listagem 2: Implementação paralela do algoritmo Couting Sort. Fonte: Implementado pelo autor.**

O algoritmo em si é quase similar ao pseudocódigo da Listagem 1, sendo que a única diferença entre os dois é que na linha 13 da Listagem 2 o índice do vetor B é subtraído de 1. Isso deve ocorrer porque vetores nas linguagens C/C++ são baseados em índice zero, isso é, o primeiro elemento encontra-se na posição 0, o segundo na posição 1 e assim por diante.
Como se pode observar, na linha 0 da Listagem 2 a função *couting_sort_parallel* recebe três argumentos, ou seja, recebe três vetores de inteiros sem sinal. Na realidade a sintaxe de vetores utilizada em argumentos de funções é apenas um "açúcar sintático" para ponteiros em C/C++.

Na linha 1 da Listagem 2 a diretiva *pragma parallel* é utilizada para indicar a região de código que será paralelizada, sendo tal região circuncidada pelo abrir e fechar de chaves. Tal diretiva é seguida *num_threads* onde *NUM_THREADS* é uma constante definida nesta pesquisa para a realização dos testes, onde seu valor vai variando de teste para teste. Os vetores *A*, *B* e *C* são compartilhados entre as *Threads* de execução, e os laços *for* das linhas 4, 8 e 12 são paralelizados pela OpenMP através da indicação da diretiva *pragma omp for* presente nas linhas 3, 7 e 11.

As constantes *MAX_INTEGERS_TO_SORT* e *MAX_INTEGER_RANGE* são definidas para a realização dos testes e variam de teste para teste. A primeira constante define o tamanho de vetor A e B e a segunda define o intervalo de valores de A e o tamanho do vetor C. Desta forma C tem o maior tamanho possível considerando-se o pior caso.

Para realizar o cálculo de *speed-up* o tempo de execução da versão paralelizada do algoritmo é contabilizado através da função *omp_get_wtime()* fornecida pela API OpenMP. De acordo com a IBM (2019) essa função retorna o número de segundos do valor inicial do tempo de *clock* real do processador, em ponto flutuante de precisão dupla. Desta forma a função é invocada duas vezes e o tempo de execução é dado pela diferença do tempo da segunda chamada pelo tempo da primeira.

De forma análoga a versão serial é contabilizada através da função *clock* definida no arquivo cabeçalho *time.h* da biblioteca C. De acordo com a página manual desta função, o valor retornado é

p tempo de CPU utilizado até ao momento e para obter os segundos dessa utilização, o valor retornado deve ser dividido pela constante *CLOCKS_PER_SEC* também definida no mesmo arquivo cabeçalho. Igualmente a versão paralelizada o tempo de execução total da porção de código de interesse é dada pela diferença das duas chamadas a função em questão.

A subseção seguinte explica sobre como é realizado o cálculo de speedup.

### 2.6. Speedup

Diversas métricas de desempenho podem ser utilizadas para verificar o ganho de desempenho sobre uma implementação serial de um problema. De acordo com Sena e Costa (2008) uma das principais métricas de desempenhos é o fator *speed-up* S(n), que representa o ganho de velocidade de processamento de uma aplicação quando executada com *n* processadores. Quanto maior for o *speed-up*, mais rápido se encontra o código paralelo. A equação que define essa métrica de acordo com Sena e Costa (2008) é dada pela seguinte equação:

$$S_n = \frac{T_s}{T_n} \tag{1}$$

Onde $T_s$ é o tempo de execução do algoritmo serial e $T_n$ o tempo de execução do algoritmo paralelo.

Sena e Costa (2008) salientam que por maior que seja a disponibilidade de processadores a serem utilizados, o fator *speed-up* é limitado por um valor máximo, decorrente da parcela serial do código. Esse valor máximo é ilustrado pela lei de Amdahl.

### 2.7. Lei de Amdahl

Nem todos os trechos de um código podem ser paralelizáveis. Neste sentido Sena e Costa (2008) explicam que se pode sempre identificar, dentro de um código, uma região que sempre será executada em serial e uma outra passível de paralelização, onde o aumento do número de processadores influenciará unicamente no tempo necessário para executar a região que pode ser paralelizada.

Segundo Sena e Costa (2008) a lei de Amdahl afirma que existe sempre um limite ao qual a capacidade de ganho pela paralelização estará sujeita. Isso ocorre devido a fatores como entrada e saída, dependência entre dados e outras questões relacionadas à aplicação e a técnica de programação paralela utilizada.

Tal limite comentado acima é de acordo com Sena e Costa (2008) calculado com base na equação 2:

$$S_p = T\frac{1}{S+(1-S)/n} \tag{2}$$

onde

$$\lim_{n \to \infty} S_p = \frac{1}{S} \quad (3)$$

Sendo $S$ — Fração serial do código e $n$ — o número de processadores.

## 3. Testes e Resultados

Os testes foram realizados em um computador pessoal, configurado com um processador Intel(R) Core(TM) 2 Duo T 8100 2.10 GHZ, totalizando dois elementos de processamento e 4 GiB de memória RAM DDR2 sobre os quais operando o sistema operacional Ubuntu 18.04.2 LTS. As implementações foram compiladas com o compilador GCC versão 7.4.0.

Os testes foram realizados com a execução do algoritmo *Couting Sort* em sua versão serial com 10 repetições; e sua versão paralela com 10 repetições para cada grupo de *threads* de tamanhos 2, 4, 8, 16, 32, 64 e 128 com dois grupos de entrada de dados: vetor aleatório e vetor inverso, onde os tamanhos definidos para os vetores são de: 10.000, 100.000, 1.000.000 e 10.000.000. Ainda cabe salientar que para os vetores aleatório e inverso de tamanho 10.000.000 utilizou-se um grupo de *threads* maior onde cada *thread* como já observado indica uma potência de 2 e para esse tamanho de vetor o número de *threads* varia até 2048 unidades. Esse conjunto de *threads* a mais se deu porque se pode observar que quanto mais se aumentava o número de *threads* também aumentavam ligeiramente as velocidades de execução do algoritmo paralelo.

| ***Taxa Speed-up - Matriz Aleatória*** | | | | |
|---|---|---|---|---|
| | ***Tamanho*** | | | |
| ***Threads*** | **10.000** | **100.000** | **1.000.000** | **10.000.000** |
| 2 | 0.40510416964 | 1.17818339566 | 1.49012424940 | 1.34362709568 |
| 4 | 1.17539401908 | 1.54182274844 | 1.43881560354 | 1.46305520694 |
| 8 | 1.06010747978 | 0.80952004606 | 1.89485083398 | 1.48170811604 |
| 16 | 0.63068574561 | 0.92796554232 | 1.81235882931 | 1.48142665312 |
| 32 | 0.29003617231 | 0.88013905460 | 1.96399972623 | 1.50256781625 |
| 64 | 0.17610180095 | 0.76768896712 | 1.97919331457 | 1.55269192217 |
| 128 | 0.09898594893 | 0.46983058577 | 1.97220163871 | 1.61225289962 |
| 256 | | | | 1.63409488302 |
| 512 | | | | 1.69786192606 |
| 1024 | | | | 1.74194542219 |
| 2048 | | | | 1.66064806996 |

**Tabela 2: Taxa de Speed-up para vetores com inteiros aleatórios.**

Através da tabela 2 pode-se observar que a maior taxa de *Speed-up* foi obtida para um vetor de tamanho 1.000.000 com 64 e 128 *Threads,* a partir dessa quantidade a taxa decrescia. Essa taxa é reflexo dos benefícios obtidos pela paralelização para tamanhos de vetores iguais ou superiores a 1.000.000. No cado do vetor com tamanho 10.000.000 pode-se obter bons resultados para além de 128 *Threads,* onde o melhor resultado obtido foi com 1024 *Threads* e decaindo acima desse valor.

Os benefícios da paralelização para vetores menores como no caso de 10.000 e 100.000

mostraram-se vantajosos apenas para pequenos grupos de *Threads*. Para um vetor de tamanho 10.000 os melhores resultado soram obtidos com 4 e 8 *Threads*. Já no vetor de 100.000 inteiros com 2 e 4 *Threads*. Acima dessas quantidades de Threads o desempenho do algoritmo apenas decaia tornando-se pior que o desempenho do algoritmo serial.

A Figura 5 ilustra a curva de s*peed-up* para as configurações descritas pela Tabela 2 de 10.000, 100.000 e 1.000.000. E a Figura 6 a curva para a configuração de 10.000.000

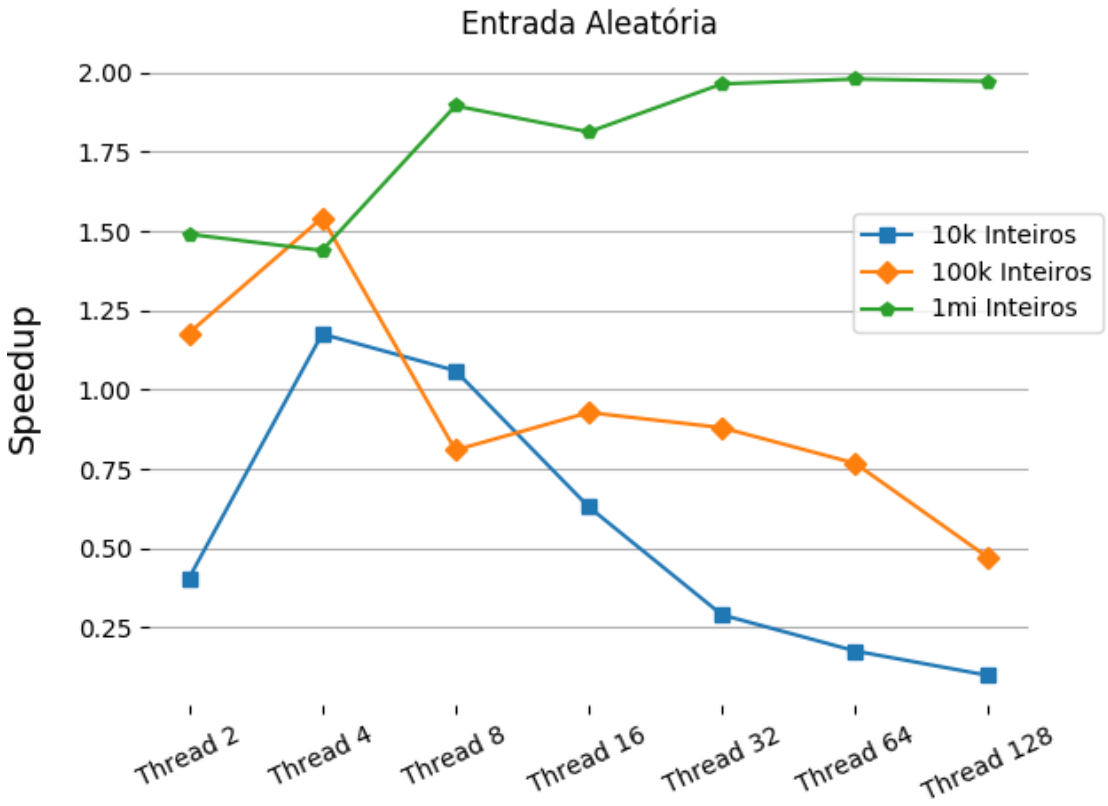


**Figura 5: Curvas das configurações da Tabela 2. Fonte: Elaborado pelo autor.**

Através da Figura 5 é possível observar mais claramente as características dos *speed-ups* obtidos. Para os vetores com tamanhos de 10.000 e 100.000 posições com valores inteiros aleatórios as curvas são crescentes apenas ao se utiliza entre de 2 a 4 *Threads* e decrescente com alguns desvios para quantidades de *Threads* maiores que 4.

Por outro lado, o vetor de 1.000.000 de inteiros os benefícios da paralelização ficam evidentes. A taxa de *speed-up* nesta configuração é bem superior as demais taxas do gráfico, além disso a curva é crescente em quase todo o seu domínio e para quantidades a partir de 32 *Threads* verifica-se que o speed-up alcançado é quase que proporcional ao número de processadores do computador utilizado neste experimento.

A curva plotada na Figura 6 representa o *speed-up* para 10.000.000 de inteiros aleatórios. Ao observá-la é possível notar que os ganhos são ainda mais notáveis nessa configuração; A curva é crescente para todas as quantidades de *Threads* as quais o algoritmo foi submetido, porém despencando a partir da quantidade de 2048 *Threads* logo depois do melhor *speed-up* obtido de aproximadamente 1.75x para 1024 *Threads*, ficando um pouco abaixo das taxas obtidas com a

ordenação do vetor de tamanho de 1.000.000 de inteiros aleatórios.

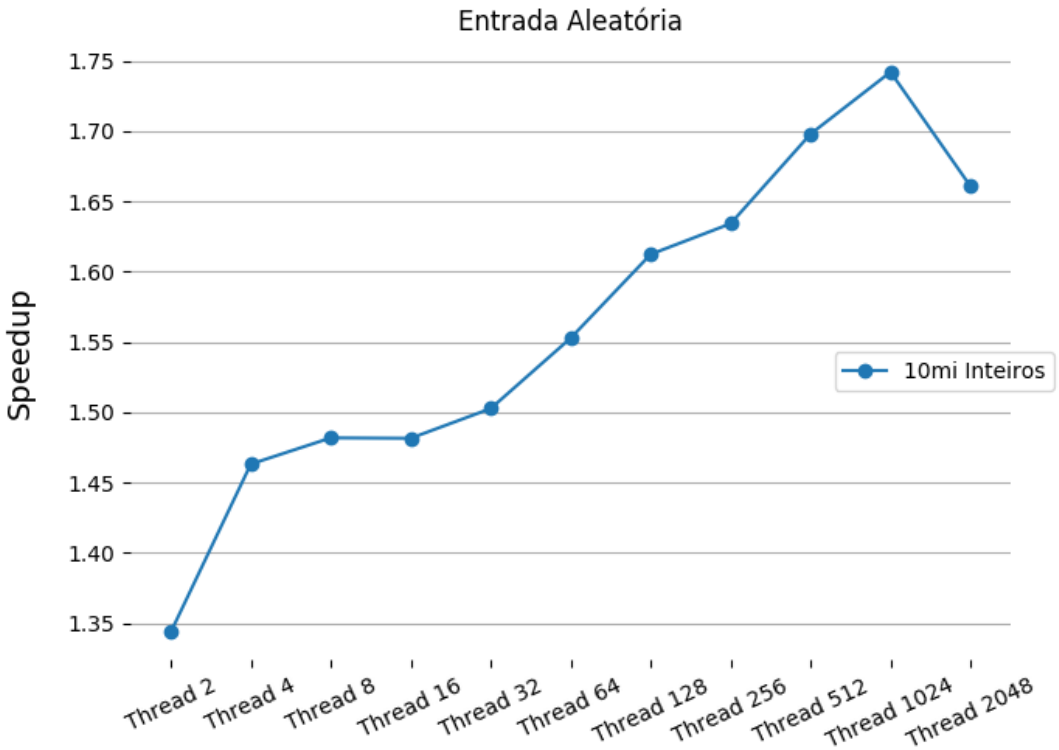


**Figura 6: Curva da configuração de 10.000.000 inteiros aleatórios da Tabela 2. Fonte: Elaborado pelo autor.**

A Tabela 3 a seguir exibe as taxas de *speed-up* para vetores de inteiros ordenados inversamente com as mesmas configurações de quantidades de *Threads* e do tamanho dos vetores da Tabela 2.

| ***Taxa Speed-up - Matriz Inversa*** | | | | |
|---|---|---|---|---|
| | ***Tamanho*** | | | |
| ***Threads*** | **10.000** | **100.000** | **1.000.000** | **10.000.000** |
| 2 | 1.0866154954286 | 0.7952875123002 | 1.4627466084649 | 1.1396771975536 |
| 4 | 0.8778443370293 | 0.8064849261549 | 1.3536043974321 | 1.1958859581836 |
| 8 | 0.4253574628698 | 1.1232103805724 | 1.1444298609809 | 1.2797592682143 |
| 16 | 0.5193461181042 | 0.9472216701544 | 1.2423263269230 | 1.3698732029839 |
| 32 | 0.3152289287600 | 0.8897997957323 | 1.0783014701242 | 1.3138819818767 |
| 64 | 0.0958696889595 | 0.6350310471379 | 1.0791662193820 | 1.3601936673706 |
| 128 | 0.0723084297102 | 0.4891828537704 | 1.1043012177037 | 1.4736529327811 |
| 256 | | | | 1.4220555497857 |
| 512 | | | | 1.3886527387052 |
| 1024 | | | | 1.0967104988229 |
| 2048 | | | | 1.1026045003693 |

**Tabela 3: Taxa de Speed-up para vetores com inteiros com ordenação inversa.**

Os benefícios da paralelização são menos visíveis para as configurações da tabela 3. Como se pode observar os vetores de tamanhos 10.000 e 100.000 mostram desvantagens ao serem paralelizados, pois o *speed-up* obtido para esses tamanhos apenas decaí mostrando um rendimento do algoritmo paralelo muito pior que do serial.

Em contrapartida para os tamanhos de 1.000.000 e 10.000.000 a paralelização mostra-se um pouco mais vantajosa. Para o vetor de 1.000.000 inteiros, os melhores *speed-up* se mostraram para 2 e 4 *Threads* a partir daí a taxa decaí, no entanto mostrando-se ligeiramente superior a versão serial. No caso de 10.000.000 inteiros o *speed-up* vai aumentando conforme o número de *Threads* aumentam obtendo-se as melhores taxas para 128 e 256 *Threads*.

As curvas dos vetores de tamanhos 10.000, 100.000 e 1.000.000 são plotadas na Figura 7 e a curva para o vetor de tamanho 10.000.000 na Figura 8.

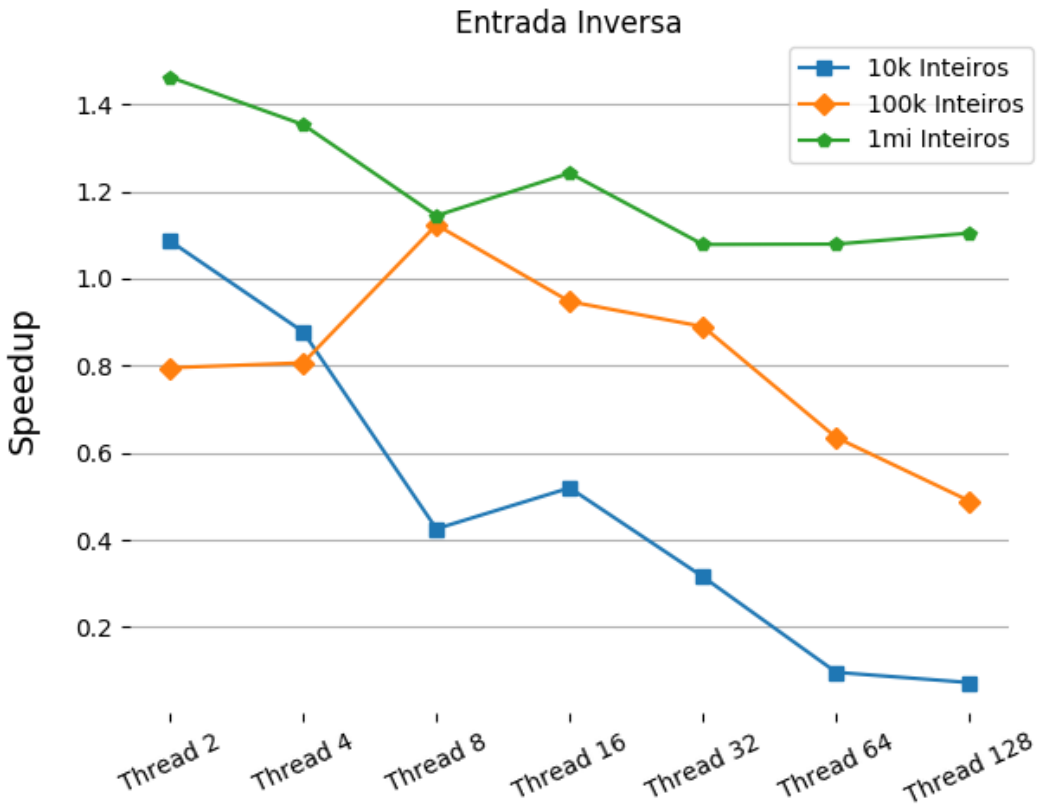


**Figura 7: Curvas das configurações da Tabela 3. Fonte: Elaborado pelo autor.**

As curvas da Figura 7 ilustram as desvantagens do paralelismo para vetores com ordenação inversa. Para o menor vetor com tamanho de 10.000 inteiros a curva decaí de forma rápida e mostra que o desempenho de paralelização que se obtêm nesta configuração é muito pior que o desempenho da execução serial, mostrando-se 10x mais lento para grandes quantidades de *Threads* tais como 64 e 128 unidades. A vantagem mostrada pelo vetor de 100.000 inteiros é de apenas 0.12x mais rápido que a versão serial para quantidade de 8 *Threads*, decaindo para quaisquer outras quantidades.

Por outro lado para o vetor de 1.000.000 de inteiros mostra-se vantajoso para configurações com 2 e 4 *Threads* onde obtêm um *speed-up* de aproximadamente 1.46x para 2 *Threads* e 1.35x para 4 *Threads*. O algoritmo paralelo mantêm-se ligeiramente mais rápido que o serial para as outras configurações de *Threads.*

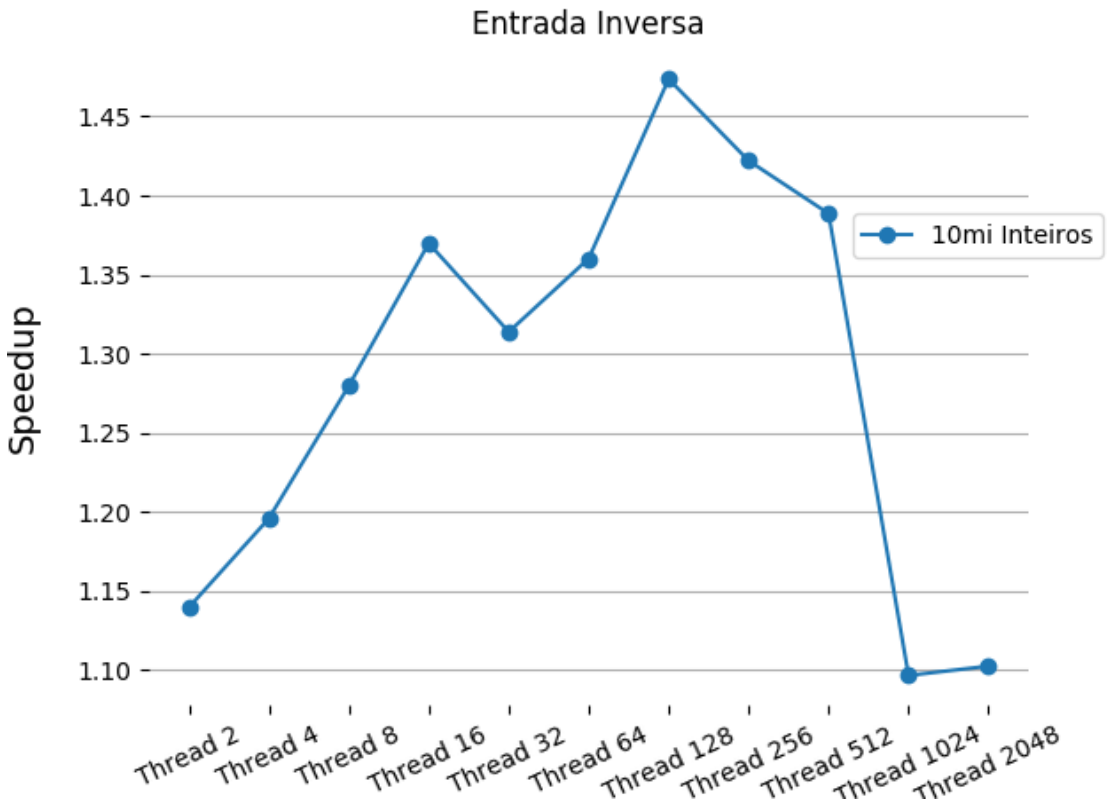


**Figura 8: Curva da configuração de 10.000.000 inteiros ordenados inversamente da Tabela 3. Fonte: Elaborado pelo autor.**

A configuração de 10.000.000 ilustrada pela Figura 8 é crescente para quase todos os tamanhos de Threads, sendo decrescente para Threads com tamanhos superiores a 128 *Threads.* As melhores taxas de *speed-up* são de 128 e 256 Threads com taxas de aproximadamente 1.47x e 1.42x respectivamente. Tais taxas semelhantemente próximas das obtidas na configuração de 1.000.000 de inteiros, porém neste caso o desempenho desta configuração é bem superior e isso é notável ao levar-se em consideração que está configuração é 10 vezes maior que a primeira bem como a taxa de *speed-up* vai aumentado conforme se aumenta o número de *Threads.*

Embora um vetor desta magnitude o algoritmo paralelo tenha se mostrado superior, as suas vantagens decaem totalmente ao se aumentar o número de *Threads* de 512 para 1024 *Threads*.

## 4. Conclusões

Através desta pesquisa pode-se constatar que é possível se obter vantagens na paralelização do algoritmo *Couting Sort*. Porém essas vantagens apenas se mostraram para vetores de grandes proporções, tais como os vetores de 1.000.000 e 10.000.000 de inteiros, o que destaca a afirmação de alguns autores consultados neste trabalho ao afirmarem que o algoritmo paralelo mostra se vantajoso para vetores de grandes tamanhos.

As principais vantagens da paralelização do algoritmo foram obtidas no caso médio, ou seja, para o caso em que os números ordenados foram gerados aleatória mente, onde os melhores resultados foram as taxas de *speed-up* de aproximadamente 2.0x e de 1.75x para os respectivos

tamanhos de vetores supracitados.

Já para os vetores ordenados de forma inversa o desempenho da implementação paralela foi pior do que a da implementação serial para vetores de tamanhos 10.000 e 100.000, sendo que *speed-ups* com taxas menores já eram esperados dado que o vetor ordenado inversamente exibe o pior caso de ordenação para algoritmos de ordenação. Por outro lado para os demais tamanhos a versão paralela mostrou-se razoavelmente superior a versão serial.

No geral o desempenho da implementação paralela do algoritmo Couting Sort se mostrou superior a implementação do mesmo em sua versão serial. Mesmo com resultados razoáveis vale salientar que o computador utilizado nesta pesquisa é antigo e possuí pouca memória, o que acaba limitando testes que poderiam ser realizados para vetores ainda maiores.

A literatura mostra através de evidências que com configurações mais robustas e processamento via GPU é possível obter taxas de *speed-up* extremamente altas, tais como por exemplo de 137x obtida por Faujdar e Ghrera (2016).

Desta forma para trabalhos futuros propõe-se o seguinte: Realizar testes em computadores mais robustos; comparar os desempenhos entre computadores diferentes porém com hardwares similares e além do processamento via CPU também realizar o processamento através de GPUs. E por último também é importante ressaltar que ferramentas de *profiling* (perfilamento) fornecem melhores resultados na mensuração de desempenhos de algoritmos, inclusive no que diz respeito a granularidade de dados e portanto neste sentido a adoção dessas ferramentas em trabalhos futuros também faz se necessário para complementar a pesquisa com resultados mais refinados.

## 5. Referências

ABELSON, H; SUSSMAN, G. J; SUSSMAN, J. **Structure and interpretation of computer programs /** Harold Abelson and Gerald Jay Sussman, with Julie Sussman. – 2nd ed. p. com. – (Eletrical engineering and computer science series), 1996 The Massachusetts Institute of Technology. ISBN-13 978-0262-01153-2

CHANDRA, Rohit; MENON, Ramesh; DAGUM, Leo; KOHR, David; MAYDAN, Dror; MCDONALD, Jeff. **Parallel Programming in OpenMP.** Elsevier, 2000. ISBN 0080513530, 9780080513539.

CORMEN, T. H; LEISERSON, C. E; RIVEST, R. L; Stein, C. **Introduction to Algorithms.** 3rd ed., no. 2. London: The MIT Press Cambridge, Massachusetts, 2009.

DUVANENKO, Victor J. **Parallel Counting Sort.** Disponível em: <http://www.drdobbs.com/architecture-and-design/parallel-counting-sort/224700144?pgno=1/>. Publicado em: 28 de Abril de 2010. Acesso em 16 de Maio de 2018 às 18:32.

FAUJDAR, Neetu; GHRERA, SatyaPrakash. **Performance Evaluation of Parallel Count Sort using GPU Computing with CUDA**. Indian Journal of Science and Technology, [S.l.], may. 2016. ISSN 0974 -5645. Disponível em: <http://www.indjst.org/index.php/indjst/article/view/80080/>. Acesso em: 16 de Maio de 2019 às 22:04. doi:10.17485/ijst/2016/v9i15/80080.

FLYNN, M. J. **Some Computer Organizations and their Effectiveness.** IEEE Transaction on Computers 21, v. 21, n. 9, pp. 948-960. (1972).

IBM. **omp_get_wtime().** Disponível em: <https://www.ibm.com/support/knowledgecenter/en/SSGH4D_16.1.0/com.ibm.xlf161.aix.doc/proguide/ompgetwtime.html/> Acesso em: 01/06/2019 às 14:09.

KAUL, Aishwarya. Article: **Space Optimization of Counting Sort**. IJCA Proceedings on International Conference on ICT for Healthcare ICTHC 2015(1):7-11, April 2016. Disponível em: <https://www.ijcaonline.org/proceedings/icthc2015/number1/24652-8250/>. Acesso em: 16 de Maio de 2019 às 22:04.

NAVAUX, P. O. A. et al., **Execução de Aplicações em Ambientes Concorrentes.** Instituto de informática - Universidade do Rio Grande do Sul - Gramado - ERAD (2001).

OPENMP. **About us.** Disponível em: <https://www.openmp.org/>. Acesso em: 16 de Maio de 2019 às 22:48.

ROSE, C. A. F; NAVAUX, P. O. A. **Fundamentos de Processamento de Alto Desempenho.** ERAD - Pelotas. (2004).

SAIAN, Pratyaksa. (2018). **Parallel Counting Sort: A Modified of Counting Sort Algorithm. International Journal of Information Technology and Business.** 1. 10-15. 10.24246/ijiteb.112018.10-15.

SENA, Maria Cecília Rodrigues; COSTA, Joseaderson Augusto de Caldas. **TUTORIAL OpenMP C/C++.** Maceió, março de 2008.

SILBERSCHATZ, A; GALVIN, P. B; GAGNE, G. **Operating System Concepts. 9 ed.** John Wiley & Sons, Inc. 111 River Street, Hoboken. Nova Jersey, Estados Unidos. 2008. ISBN 978-1-118-06333-0 (2008).