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

STRAND SORT PARALELO

Matheus Giacomelli Dos Santos[1], Rodrigo Curvêllo[2]
[1]Aluno Graduando Em Ciência Da Computação
[2]Professor Orientador Do Instituto Federal Catarinense
[1]matheusgds66@gmail.com, [2]rodrigo.curvello@ifc.edu.br

Abstract. *This article presents about the sort method known as Strandsort, with emphasis on parallel programming. And also discussing the relationship of programming in parallel and serial, thus showing how best to use this method of ordering.*
Key-words: *Parallel Programming; Data Structure; Method Of Ordering.*

Resumo. *Este artigo apresentara sobre o método de ordenação conhecido como Strandsort, com ênfase em programação paralela. E também discutindo sobre a relação da programação em paralela e em serial, assim apresentando qual a melhor forma de utilizar este método de ordenação.*
Palavras-chave: *Programação Paralela; Estrutura De Dados; Métodos De Ordenação.*

1. Introdução

Na ciência da computação, existem dados, e agora com avanço dos sistemas, surge a BIG DATA, que é uma base enorme desses dados e estes são necessários ser ordenados para uma melhor pesquisa e até mesmo para um melhor desempenho e para isto podemos levar conta os métodos de ordenação, existem métodos para todas as ocasiões cada um com sua vantagem e desvantagem levando em consideração o melhor caso, pior caso, e o caso médio, através disto se obtém uma melhor velocidade de resposta pelo método ordenação. Existem métodos de ordenação mais conhecidos e menos conhecidos, os mais conhecidos são: Inserction Sort, Selection Sort, Bubble Sort, Quick. E inúmeros outros menos conhecidos. Neste artigo será apresentado o método de ordenação Strand Sort, um método menos conhecido. Não se sabe ainda ao certo sobre a história, não sabe se o ano nem o autor, entretanto segundo a empresa NIST acredita-se numa hipótese que surgiu pouco tempo atrás, através de um e-mail que surgiu em novembro de 1997, o autor ao criar o algoritmo teve como ideia criar um algoritmo rápido com poucos dados onde poderia levar o melhor caso possível, ou seja, melhor caso seria O(N) onde alguns dados já estariam ordenados, já no pior caso poderia levar O (N*N) e o autor também levou em consideração o uso da memoria em relação ao desgaste. O método foi criado com o intuito serial, entretanto pode se usar em processamentos paralelos, o autor tentou melhorar o algoritmo comparando primeiro elemento e ultimo elemento, comparando de trás pra frente, de frente pra trás e acreditava que não compensava em relação a velocidade.

2. Metodologia

2.1. Estrutura De Dados

Segundo Laureano (2008), as estruturas de dados formam um tipo de armazenamento dos tipos de dados para tratamento dessas informações, e se baseiam em armazenamentos vistos no dia a dia, são estruturas já conhecidas e já utilizadas no mundo real e adaptadas para o mundo computacional, por isso cada tipo de estruturas de dados possui vantagens e desvantagens e cada uma delas tem sua área de atuação otimizada.

2.2. Ordenação

Segundo Laureano (2008), algoritmo de ordenação em ciência da computação é um algoritmo que coloca os elementos de uma dada sequência em certa ordem – em outras palavras, efetua sua ordenação completa ou parcial. As ordens mais usadas são a numérica e lexicográfica. Existem várias razões para se ordenar uma sequência, uma delas é a possibilidade de acessar seus dados de modo mais eficiente.

2.3. OPENMP

O programador utiliza diretivas de compilação inseridas no código sequencial para informar ao compilador quais as regiões devem ser paralelizadas. Com base nestas diretivas o compilador gera um código paralelo. Dessa forma é necessário um compilador "especial", ou seja, capaz de entender as diretivas e gerar o código multi-thread. Pode-se citar o padrão OpenMP como uma interface que especifica um conjunto de diretivas de compilação, funções e variáveis de ambiente para a programação paralela em arquiteturas com memória compartilhada.

2.4. Programação Concorrente e Programação Paralela

Segundo Andrei Alvares (2015) e Tiago Ferreto (2015), Programação Concorrente é o termo usado em computação para designar situações nas quais diferentes processos competem pela utilizam de algum recurso, cooperam para a realização de uma mesma tarefa. Já Programação Paralela é a divisão de uma determinada aplicação em partes, de maneira que essas partes possam ser executadas simultaneamente, por vários elementos de processamento, e os elementos de processamento devem cooperar entre si, utilizando primitivas de comunicação e sincronização, realizando a quebra do paradigma de execução sequencial do fluxo de instruções.

2.5. Método StrandSort

Conforme o Instituto Nacional de Padrões e Tecnologia, "Um algoritmo de *classificação* que funciona bem se muitos itens estiverem em ordem. Primeiro, inicie uma sub-lista movendo o primeiro item da lista original para a sub-lista. Para cada item subsequente na lista original, se for maior que o último item da sub-lista, remova-o da lista original e anexe-o à sub-lista. *Mesclar* a sub-lista em uma lista final e

classificada. Repetidamente extrair e mesclar sub-listas até que todos os itens sejam classificados. Manuseie dois ou menos itens como casos especiais".

Exemplo 1:

→	5	1	4	2	0	9	6	8	3	7
→	5	9								
→	5	9								
→	1	4	2	0	6	8	3	7		
→	1	4	6	8						
→	1	4	5	6	8	9				
→	2	0	3	7						
→	2	3	7							
→	1	2	3	4	5	6	7	8	9	
→	0									
→	0									
→	0	1	2	3	4	5	6	7	8	9

LEGENDA:

→ LISTA INICIAL
→ SUB-LISTA
→ LISTA SAÍDA

Fonte: http://algolab.valemak.com/strand. Modificado pelo autor

Exemplo 2 :

Lista Não Ordenada	Sub-lista	Lista Ordenada
3 1 5 4 2		
1 4 2	3 5	
1 4 2		3 5
2	1 4	3 5
2		1 3 4 5
	2	
		1 2 3 4 5

Fonte: https://www.youtube.com/watch?v=t8r6wyG8eAU . Modificado Pelo Autor

2.6 Speedup – Métrica de Desempenho

Segundo Rocha (2007), as principais métricas de avaliação de desempenho de aplicações paralelas são: Speedup, Eficiência, Redundância, Utilização e Qualidade.

A métrica principal, das citadas a cima, é o fator speedup, Equação 1. Esse fator representa o ganho de velocidade de processamento de uma aplicação quando executada com n processadores. Quanto maior o speedup, mais rápido se encontra o código paralelo.

Segundo Orellana (2011) o speedup (Equação 1) é definido como a razão entre o tempo de execução do algoritmo sequencial $TS = (n)$, que depende apenas do tamanho n do problema, e o tempo de execução do algoritmo paralelo $Tp(p, n)$, que depende do tamanho do problema e da quantidade p de processadores utilizados.

Equação 1 – Equação do Speedup

$$S(n,p) = TS(n)/\ Tp(p, n)$$

3. Resultados e Discussões

3.1 Código

```
8   #include <iostream>
9   #include<thread>
10  #include <chrono>
11
12  using namespace std;
13
14  const int dadosrand = 1000;
15  const int tamanho = 100;
16  const int numerothread = 2;
17  const int valorlooptempo = 1000;
18
19  list<int> listaEntrada;
20  list<double>::iterat
21  list<int>::iterator k;
22  list<int>::iterator j;
23  list<int>::iterator p;
```

std::list<int> listaEntrada

Fígura 01 – Variáveis Do Programa. Fonte: Próprio autor

```
41      list<int> strandSort(list<int> entrada, list<int> saida)
42      {
43
44          if (entrada.empty()) {
45              return saida;
46          }
47
48
49          list<int> sublist;
50          sublist.push_back(entrada.front());
51          entrada.pop_front();
52
53          for (auto it = entrada.begin(); it != entrada.end(); ) {
54
55
56              if (*it > sublist.back()) {
57                  sublist.push_back(*it);
58
59                  it = entrada.erase(it);
```

Fígura 02- 1 Parte Método Stand Recursivo. Fonte: Próprio autor

```
56              if (*it > sublist.back()) {
57                  sublist.push_back(*it);
58
59                  it = entrada.erase(it);
60              }
61
62              else
63                  it++;
64          }
65
66          saida.merge(sublist);
67
68          return strandSort(entrada, saida);
69      }
```

Fígura 03 – 2 Parte Método StrandSort Recursivo. Fonte: Próprio autor

```
86      for (int i = 0; i < tamanho; i++) {
87          int x = (rand() % dadosrand)+1;
88          listaEntrada.emplace_front(x);
89      }
90      printarLista(listaEntrada);
91      cout << endl;
92      auto tempinicial = std::chrono::steady_clock::now();
93
94      omp_set_num_threads(numerothread);
95      #pragma omp flush(listaEntrada)
96      {
97          listaSaida = strandSort(listaEntrada, listaSaida);
98      }
99      double tempoprocessamento = double(std::chrono::duration_cast <std::chrono::nanose
100
101     printarLista(listaSaida);
```

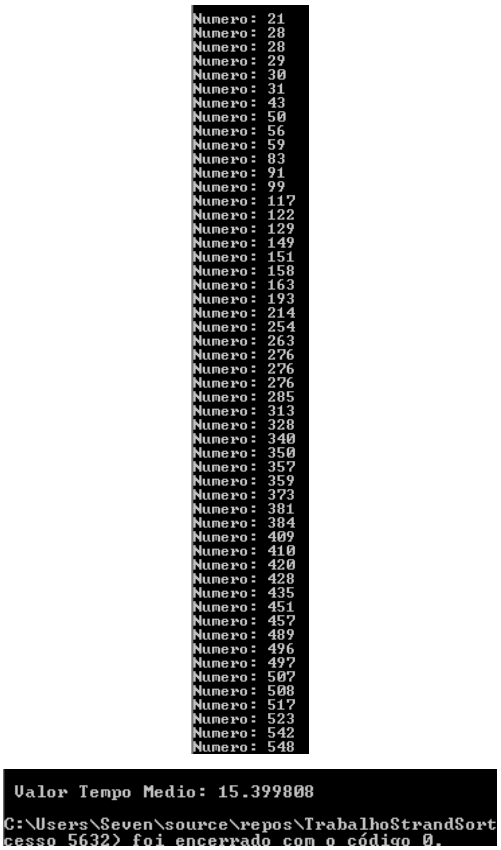
Figura 04 – Carregamento Números Randômicos e a Diretiva OPENMP.

3.2 Exemplos

```
Numero: 129
Numero: 409
Numero: 956
Numero: 636
Numero: 854
Numero: 548
Numero: 263
Numero: 619
Numero: 985
Numero: 149
Numero: 83
Numero: 574
Numero: 384
Numero: 276
Numero: 158
Numero: 313
Numero: 50
Numero: 565
Numero: 859
Numero: 674
Numero: 523
Numero: 497
Numero: 600
Numero: 855
Numero: 373
Numero: 428
Numero: 567
Numero: 507
Numero: 328
Numero: 826
Numero: 785
Numero: 28
Numero: 852
Numero: 151
Numero: 489
Numero: 496
Numero: 117
Numero: 31
Numero: 357
Numero: 602
Numero: 887
Numero: 43
Numero: 435
Numero: 847
Numero: 163
Numero: 669
Numero: 674
Numero: 726
Numero: 193
Numero: 30
Numero: 992
Numero: 791
```

- **Observação: Tempo Retirado Em Segundos.**

Figura 05 – Vetor de 100 posições com randômico 1000 na diretiva Flush. Fonte: Próprio autor

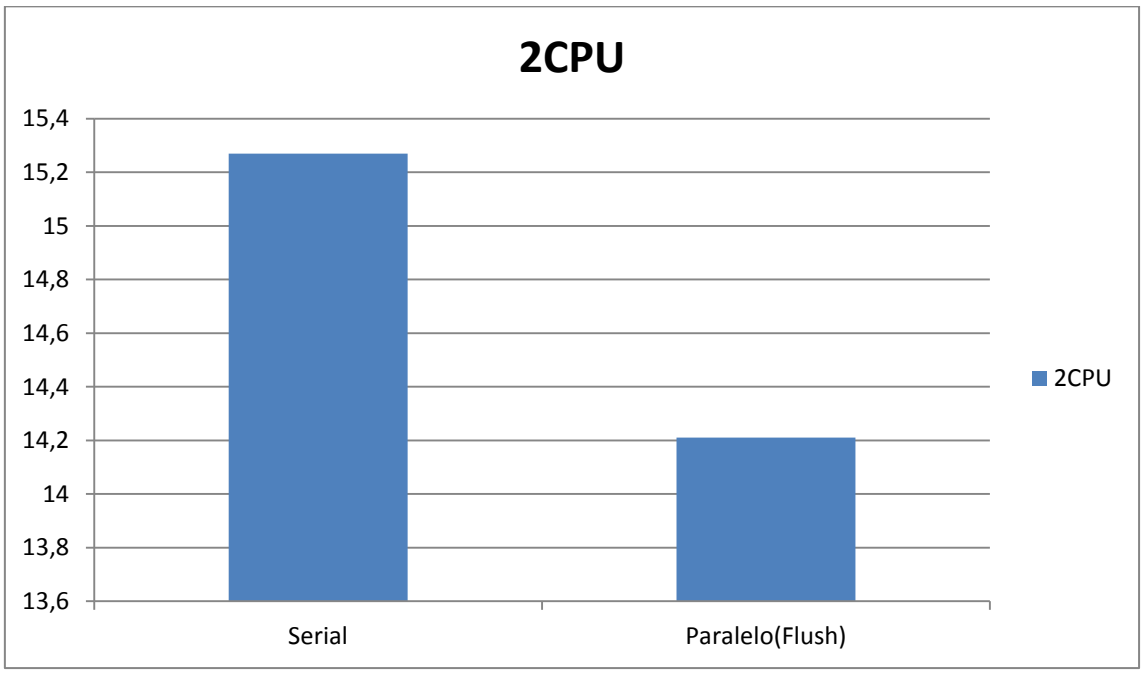


- **Observação: Tempo Retirado Em Segundos.**

Fígura 06 – Obtendo tempo de 1 vetor. Fonte: Próprio autor

3.3 Gráficos e Tabelas

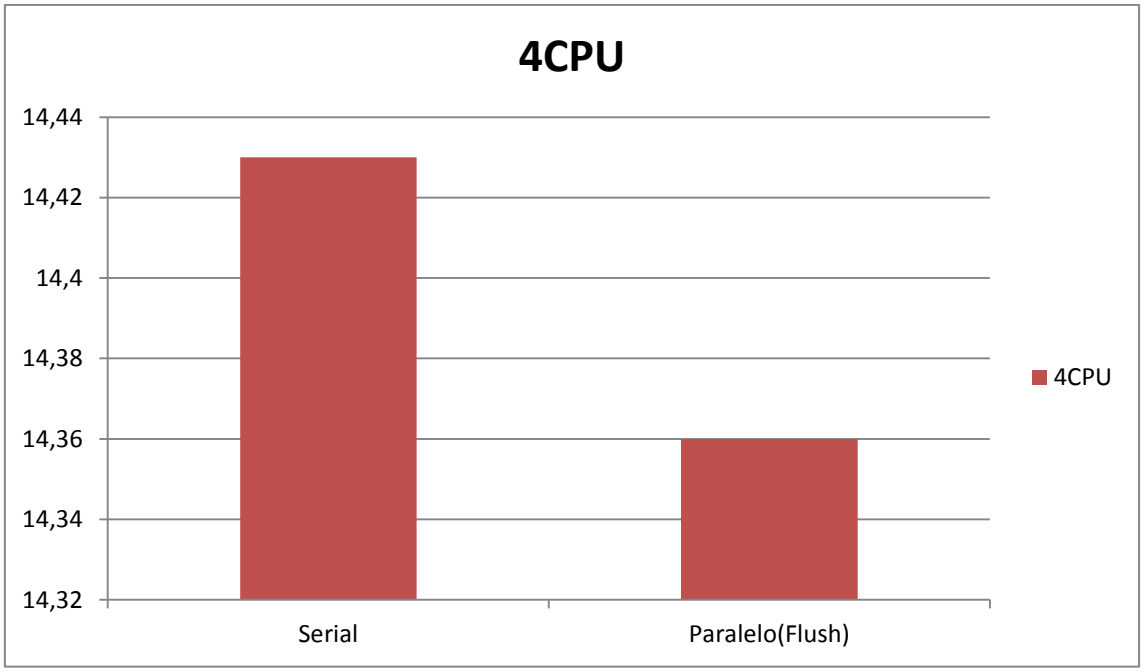
Gráfico 1- 100 Dados Rand 1000 (2CPUS)



Fígura 07- Tempo médio 1000 vetores de 100 posições com randômico de 1000 na diretiva Openmp (Flush) com 2CPU. Fonte: Próprio autor.

Fígura 08 – Tempo Medio 1000 vetores de 100 posições com randômico 1000. No Serial Com 2 CPU. Fonte: Próprio autor.

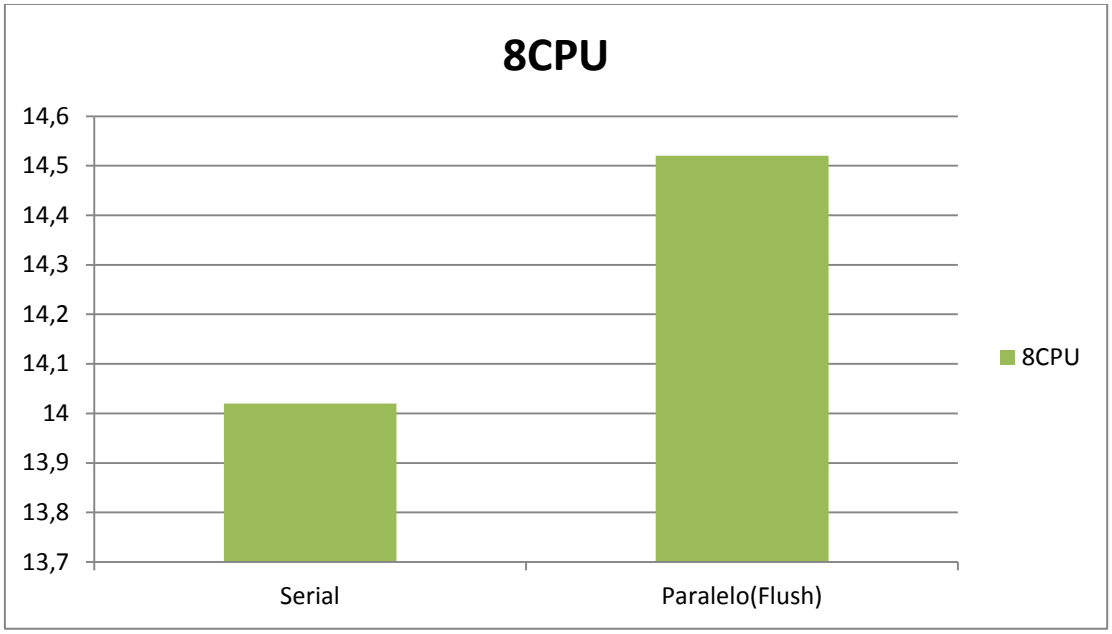
Gráfico 2- 100 Dados Rand 1000 (4CPUS)



Fígura 07- Tempo médio 1000 vetores de 100 posições com randômico de 1000 na diretiva Openmp (Flush) com 4 CPU. Fonte: Próprio autor.

Fígura 08 – Tempo Medio 1000 vetores de 100 posições com randômico 1000. No Serial Com 4 CPU. Fonte: Próprio autor.

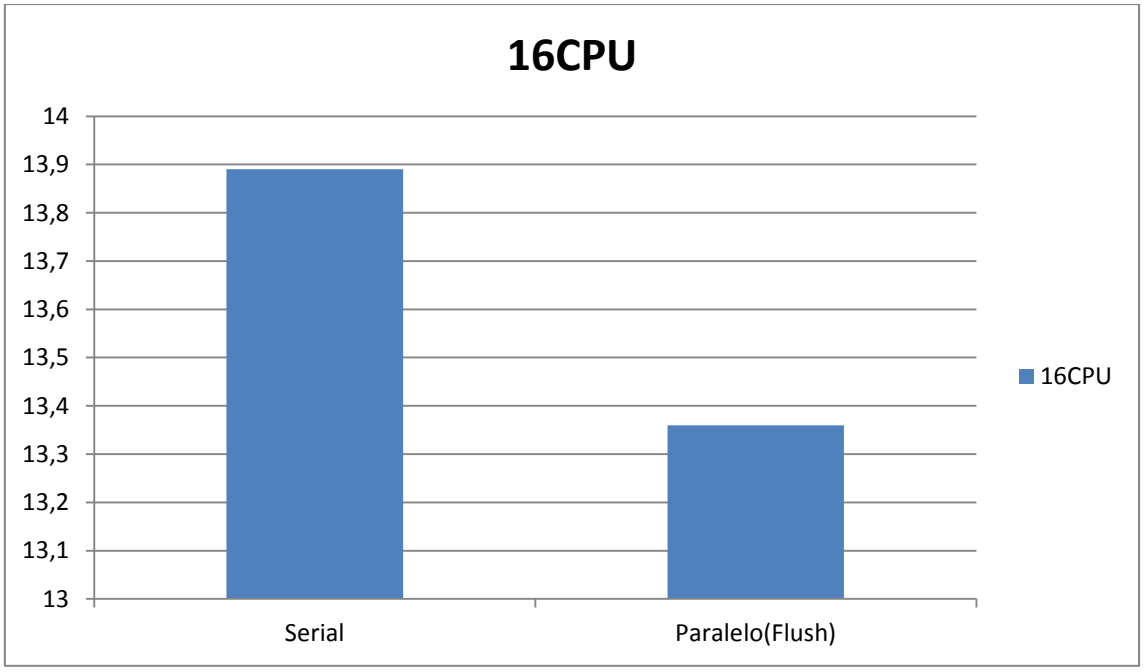
Gráfico 3- 100 Dados Rand 1000 (8CPUS)

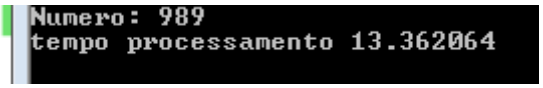


Fígura 07- Tempo médio 1000 vetores de 100 posições com randômico de 1000 na diretiva Openmp (Flush) com 8 CPU. Fonte: Próprio autor.

Fígura 08 – Tempo Medio 1000 vetores de 100 posições com randômico 1000. No Serial Com 8 CPU. Fonte: Próprio autor.

Gráfico 4- 100 Dados Rand 1000 (16CPUS)





Fígura 07- Tempo médio 1000 vetores de 100 posições com randômico de 1000 na diretiva Openmp (Flush) com 16 CPU. Fonte: Próprio autor.

Fígura 08 – Tempo Medio 1000 vetores de 100 posições com randômico 1000. No Serial Com 16 CPU. Fonte: Próprio autor.

Tabela 1- Speedup 100 Dados Rand 1000 (Média 1000 vetores)

	2 CPU	***4CPU***	***8CPU***	***16CPU***
Serial (Ts(n))	*15.27*	*14.43*	*14.52*	*13.89*
Paralelo(Tp(n,p))	*14.21*	*14.36*	*14.02*	*13.36*
Speedup(S(n,p))	*1.07*	*1*	*1.03*	*1.03*

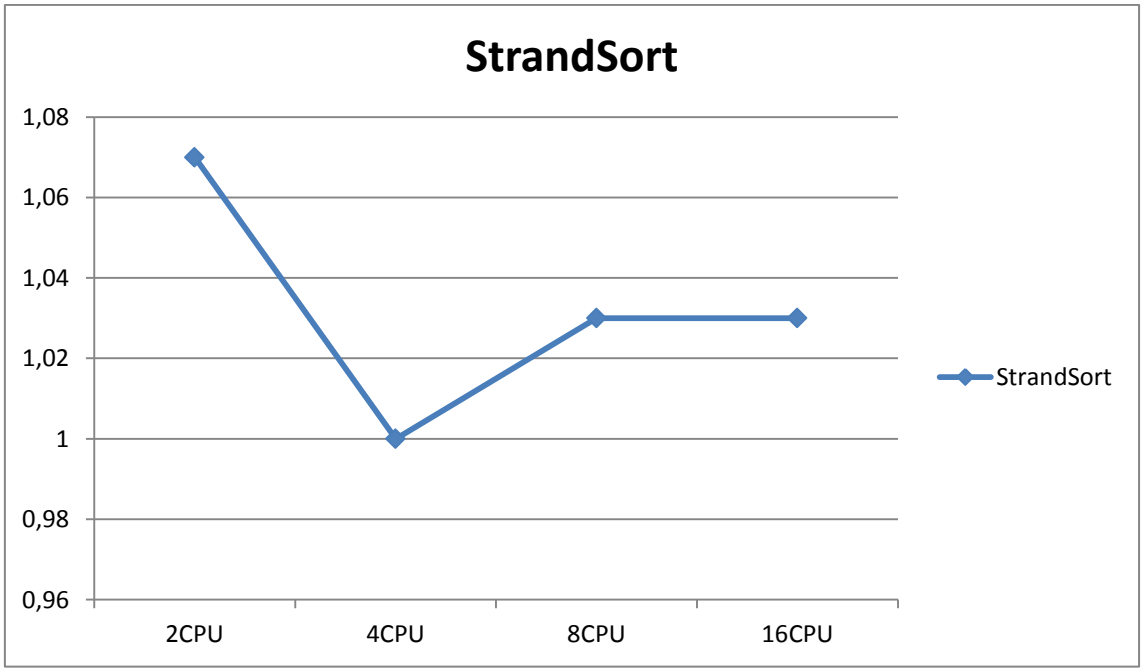


Gráfico 5 – Speedup 100 Dados Rand 1000

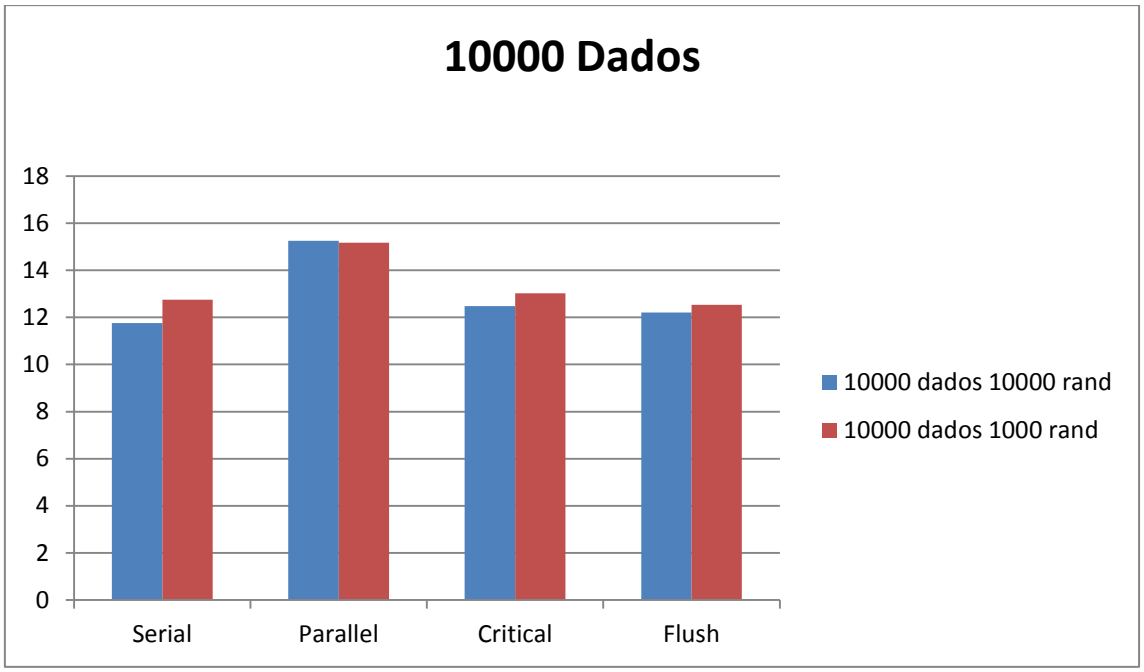


Gráfico 6 - Exemplo 10.000 Dados(2CPUS)

3.4 Pseudocódigos

3.4.1 C++

```cpp
#include <list>

template <typename T>
std::list<T> strandSort(std::list<T> lst) {
  if (lst.size() <= 1)
    return lst;
  std::list<T> result;
  std::list<T> sorted;
  while (!lst.empty()) {
    sorted.push_back(lst.front());
    lst.pop_front();
    for (typename std::list<T>::iterator it = lst.begin(); it != lst.end(); ) {
      if (sorted.back() <= *it) {
        sorted.push_back(*it);
        it = lst.erase(it);
      } else
        it++;
    }
    result.merge(sorted);
  }
  return result;
}
```

Fígura 09 – Código Strand em C++. Fonte: https://rosettacode.org/wiki/Sorting_algorithms/Strand_sort#Java

3.4.2 Java

```java
import java.util.LinkedList;

public class Strand{
	// note: the input list is destroyed
	public static <E extends Comparable<? super E>>
	LinkedList<E> strandSort(LinkedList<E> list){
		if(list.size() <= 1) return list;

		LinkedList<E> result = new LinkedList<E>();
		while(list.size() > 0){
			LinkedList<E> sorted = new LinkedList<E>();
			sorted.add(list.removeFirst()); //same as remove() or remove(0)
			for(Iterator<E> it = list.iterator(); it.hasNext(); ){
				E elem = it.next();
				if(sorted.peekLast().compareTo(elem) <= 0){
					sorted.addLast(elem); //same as add(elem) or add(0, elem)
					it.remove();
				}
			}
			result = merge(sorted, result);
		}
		return result;
	}

	private static <E extends Comparable<? super E>>
	LinkedList<E> merge(LinkedList<E> left, LinkedList<E> right){
		LinkedList<E> result = new LinkedList<E>();
		while(!left.isEmpty() && !right.isEmpty()){
			//change the direction of this comparison to change the direction of the sort
			if(left.peek().compareTo(right.peek()) <= 0)
				result.add(left.remove());
			else
				result.add(right.remove());
		}
		result.addAll(left);
		result.addAll(right);
```

Fígura 10 – Código Strand em Java. Fonte: https://rosettacode.org/wiki/Sorting_algorithms/Strand_sort#Java

3.4.3 Haskell

```haskell
-- Same merge as in Merge Sort
merge :: (Ord a) => [a] -> [a] -> [a]
merge [] ys = ys
merge xs [] = xs
merge (x : xs) (y : ys)
	| x <= y = x : merge xs (y : ys)
	| otherwise = y : merge (x : xs) ys

strandSort :: (Ord a) => [a] -> [a]
strandSort [] = []
strandSort (x : xs) = merge strand (strandSort rest) where
	(strand, rest) = extractStrand x xs
	extractStrand x [] = ([x], [])
	extractStrand x (x1 : xs)
		| x <= x1 = let (strand, rest) = extractStrand x1 xs in (x : strand, rest)
		| otherwise = let (strand, rest) = extractStrand x xs in (strand, x1 : rest)
```

Fígura 11 – Código Strand em Haskell. Fonte: https://rosettacode.org/wiki/Sorting_algorithms/Strand_sort#Java

4. Conclusão

Pode se observar que de acordo o gráfico 5 de Speedup, verificamos que em alguns casos o método strand sort funciona melhor paralelizado na diretiva OpenMP Flush, a diretiva atualiza a lista de entrada dos dados, pois o método é recursivo, outros casos ainda obtém se melhor desempenho na parte serial do método strand sort. O método em si quando criado acreditava-se que não poderia se obter melhor desempenho em paralelizável. Podemos verificar também que quanto mais núcleos trabalhamos do processador sendo este a partir dos 8 núcleos, começa-se a perder desempenho, o

método sempre depende do último elemento do vetor para verificar outros antigos elementos, isto dificulta o trabalho de paralelizamento. Com isso concluímos que podemos utilizar o método strandsort tanto serializado, quanto paralelizado, entretanto o método funciona com uma base de dados menor, para remover em ordem sequencial do vetor.

Acesso ao Projeto no GitHub: https://github.com/matheusgds/Programacao-Paralela-Multicore

5. Referências

ÁLVARES, Andrei Rimsa. **Programação Concorrente.** Minas Gerais: Andrei Rimsa Álvares, [2015]. 64 slides, color. Disponível em: <https://homepages.dcc.ufmg.br/~rimsa/documents/decom009/lessons/Aula09.pdf>. Acesso em: 13 maio 2019.

CURVELLO, Rodrigo. **PPMC:** Rio do Sul: Rodrigo Curvello, 2018. 48 slides, color.

DEITEL, Paul; DEITEL, Harvey. **Java:** Como Programar. 10. ed. São Paulo: Pearson Education, 2016. 906 p.

FERRETO, Prof. Tiago. **Programação Paralela.** Rio Grande do Sul: Prof. Tiago Ferreto, [2015]. 77 slides, color. Disponível em: <https://www.inf.pucrs.br/~gustavo/disciplinas/ppd/material/Prog_Paralela.pdf>. Acesso em: 13 maio 2019.

LAUREANO, Marcos. **Estrutura De Dados Com Algoritmos e C.** Curitiba: Brasport Livros e Multimídia Ltda., 2008. 150 p. Disponível em: <http://www.mlaureano.org/livro/livro_estrutura_conta.pdf>. Acesso em: 13 maio 2019.

NIST: NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY, **StrandSort**, 1997. Disponível em: https://xlinux.nist.gov/dads/HTML/strandSort.html. Acesso em: 26 maio 2019.

Paul E. Black, "**strand sort**", em *Dictionary of Algorithms and Data Structures* [online], Paul E. Black, ed. 24 de Novembro de 2008. Acesso em: https://www.nist.gov/dads/HTML/strandSort.html. Acesso em: 26 maio 2019.

STRAND Sort Algorithm. [s.i.]: Tech Dose, 2018. YouTube, son., color. Disponível em: <https://www.youtube.com/watch?v=t8r6wyG8eAU>. Acesso em: 26 maio 2019.