CONGRESSO CATARINENSE DE CIÊNCIA DA COMPUTAÇÃO
2018. Ano 5, Vol 5. ISSN: 2319-0655

Artigo – Aplicação do *Bubble Sort* em *OpenMP*

Kael Lucas Pinheiro Cordeiro[1], Rodrigo Curvêllo[1]
[1]Instituto Federal de Educação, Ciência e Tecnologia Catarinense (IFC) – Campus Rio do Sul 89160-202 – Rio do Sul – SC - Brasil
kaellucas@hotmail.com, rodrigo.curvello@gmail.com

Abstract. *In this article we present the theoretical details and the implementation of the algorithm of ordering method with the programming language in C/C++ with the ordering of Bubble Sort in serial and parallel. Currently, OpenMP has had the advanced performance of the computer application that it is practical of parallel processing. They have managed to exceed the features provided for the faster single processors. In the parallel programming in OpenMP, they have an adequate performance of single processor that has been enjoying a significant cost advantage that have been implemented in parallel in systems that use in multiples, lower cost, convenience in microprocessors. Applications that depend on the power of more than one processor are numerous. Clearly, parallel computing can have a huge impact on application performance, and OpenMP facilitates access to this improved performance.(Chandra, 2001).*
Key-words: *Algorithm, Ordering, OpenMP, C/C++, Bubble Sort, Performance, Processor, Application.*

Resumo. *Nesse artigo vamos apresentar o detalhamento teórico e da implementação do algoritmo de método de ordenação com a linguagem de programação em C/C++ com a ordenação do Bubble Sort em serial e paralelo. Atualmente, o OpenMP teve o desempenho avançado da aplicação do computador que ele é prático do processamento paralelo. Eles conseguiram exceder com os recursos fornecidos para os processadores únicos mais rápidos. Nas programações paralelas em OpenMP, eles tem um desempenho adequado de processador único que vem desfrutando de uma vantagem pelo custo significativo que foram implementados em paralelo em sistemas que utilizam em múltiplos, custo mais baixo, comodidade em microprocessadores. Aplicativos que dependem do poder de mais de um processador são numerosos. Claramente, a computação paralela pode ter um enorme impacto no desempenho do aplicativo, e o OpenMP facilita o acesso a esse desempenho aprimorado.(Chandra, 2001).*
Palavras-chave: *Algoritmo, Ordenação, OpenMP, C/C++, Bubble Sort, Desempenho, Processador, Aplicativo.*

1. Introdução

O presente artigo vai auxiliar em muito quem precisa escrever sua própria rotina, pois vai mostrar que o *Bubble Sort* que tem como característica de memória que ocupa

pouco espaço para armazenamento, o que permite também o suporte por vários compiladores, e, ser um algoritmo simples de ordenação de itens na memória.

Trata-se de algoritmo de muitas vantagens: “é simples de escrever, fácil de entender, e leva somente algumas linhas de código”. Para haver pouca sobrecarga na memória, ocorre a ordenação dos dados no lugar, que uma vez estando na memória estão prontos para processamento.

2. Fundamentação Teórica

2.1. *OpenMP*

O *OpenMP* é uma *API* que ela serve para uma programação paralela com as linguagens de programação em C, C++ e *Fortran*, desenvolvida e mantida pelo *OpenMP*. Disponibiliza por uma alternativa de uma forma simples e portável as soluções de mais baixo nível como *POSIX threads*, e é suportado por vários compiladores como o *GCC*.

Ele consegue distinguir em outras soluções para suporte a paralelização em memória partilhada pelo seu nível de abstração, na medida em que o essencial das suas funcionalidades é obtido através de um conjunto de diretivas do compilador que ela especifica de uma forma declarativa como é que em diferentes partes do código podem ser executadas de uma forma paralela, mas o *OpenMP* também suporta com as cláusulas e sincronizações.

Enquanto isso, as aplicações mais complexas podem também necessitar pelas chamadas as funções de mais baixo nível, e que elas tornam uma compilação da aplicação dependente do *OpenMP*. O *OpenMP* usa as *threads* para que elas possam obter em um paralelismo e que as *threads* usam o mesmo espaço de endereçamento de memória, sendo que os dados podem ser partilhados por todas as *threads*.

2.2. *Bubble Sort*

Na fundamentação do *Bubble Sort*, na ano de 1962, o Iverson é o cientista da computação que usou o nome “*Bubble Sort*” para que ele possa descrever o algoritmo. Para trabalhar com essa ordenação, a classificação das bolhas descrevem como encerrar o tipo de uma forma antecipada para que ele segue um vetor se ela esteja classificado e também adquirir a otimização dela para verificar as trocas dos elementos após as iterações com o laço de repetição do *Bubble Sort*.

As origens da popularidade do *Bubble Sort*, teve uma aparição no ano de 1971. Por enquanto, o tipo de bolha é fácil de lembrar e programar essa ordenação e com as características de desempenho dessa ordenação mostravam que teve em várias medidas,

que essa ordenação não é mais simples de codificar do que os outros tipos de ordenação e seu desempenho dele é ruim e a popularidade dele foi de uma forma inexplicável.

O funcionamento da ordenação do *Bubble Sort* é bastante simples, e talvez seja o método de ordenação mais difundido. Mas para começar a funcionar com essa iteração desse algoritmo, primeiro, ele deve limitar para percorrer uma tabela do início ao fim, sem interrupção, em seguida, ele vai trocando a posição pelos dois elementos em sequência sempre que estes se apresentam fora de ordem (SZWARCFITER, p.161).

Enquanto isso, com uma intenção desse algoritmo, esse método consegue realizar um movimento para os elementos maiores em direção ao fim da tabela. Para terminar com a primeira iteração pode-se garantir que as trocas realizadas posicionam o maior elemento na última posição. Na segunda iteração dessa ordenação, o segundo maior elemento é posicionado, e assim sucessivamente, mas o processamento dele é repetido então $n - 1$ vezes (SZWARCFITER, p.161).

Para a complexidade dessa ordenação de cada uma delas, é que no melhor caso é o O(n), em que o algoritmo do *Bubble Sort* possa executar em n operações relevantes, mas com isso o n representa o número de elementos do vetor. Já no caso médio e pior caso, acontece por O (n^2) operações.

Portanto, ainda no caso médio e pior caso, o O (n^2) ocorre devido aos percursos estipulados para as variáveis i e j. Entretanto, em alguns critérios de parada que levariam em consideração com as comparações desnecessárias, isto é, comparações executadas em partes da tabela sabidamente já ordenadas. No melhor caso, o O (n) que uma vez se a tabela já esta ordenada, em que apenas um percurso é realizado (SZWARCFITER, p.161).

Listagem 1: Pseudocódigo do *Bubble Sort.* Fonte: Acervo do autor.

```
0 public class Bubble {
1   public int[] ordenar(int[] v){
2   int j;
3   int i;
4   int aux;
5     for(i = 0; i < v.lenght; i++){
6      for(j = 0; j < v.lenght-1; j++){
7        if(v[j] > v[j + 1]){
8          aux = v[j];
9          v[j] = v[j + 1];
10         v[j + 1] = aux;
11       }
12     }
13   }
14     return v;
15  }
16 }
16              }
17        }
```

Na listagem 1, o pseudocódigo do *Bubble Sort*, com o método para ordenar a bolha é pela posição dos vetores e com o laço da repetição da ordenação da bolha segue pelo tamanho de um vetor. Mas a quantidade do número de elementos aparecem em uma saída como uma ordem decrescente que ela está completamente desordenado e agora ela deve ordenar completamente para que o número de elementos fiquem mais ordenados de uma forma crescente. A ordenação segue pelo processo de repetição de *n* – 1 vezes, para começar uma troca para ordenar os números é preciso ter um auxiliar na posição de um vetor para que ele possa ordenar completamente.

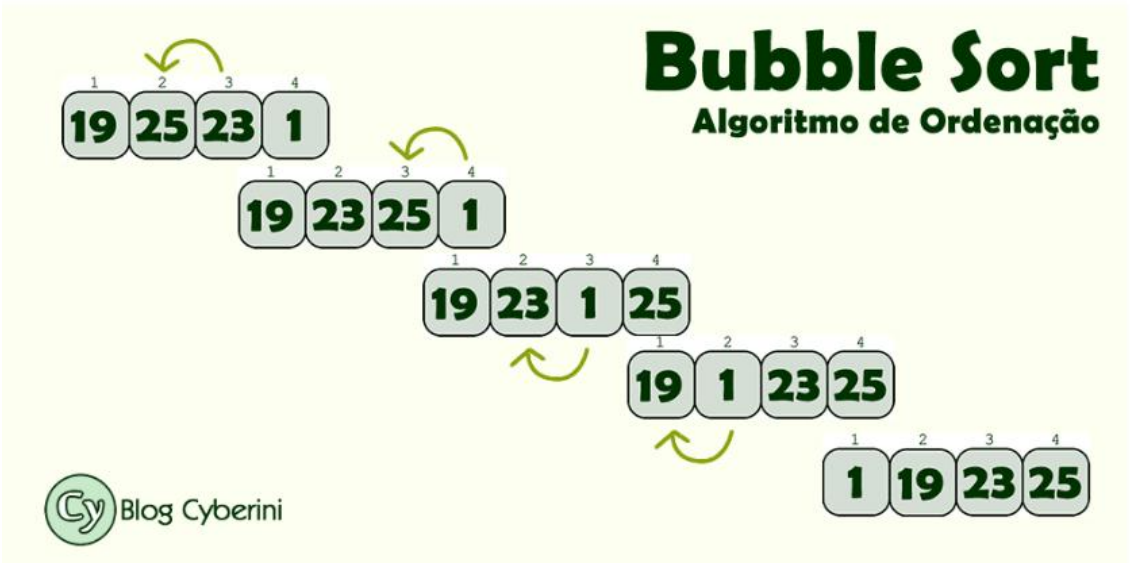

Figura 1: Demonstração da ordenação do *Bubble Sort.* Fonte: Henrique Felipe.

Na figura 1, ele mostra os passos que ele ordena o número dos elementos com a posição de um vetor para que ele acontece com a ordenação da bolha conforme o processo de uma repetição de *n* – 1 vezes.

2.3. Lei de Amdahl

A lei de Amdahl surgiu pelo projetista e empreendedor Gene Myron Amdahl, com essa lei ele consegue tomar em uma medida pelo cálculo do *Speedup* para que possamos obter uma solução de um determinado problema com o *n* processadores e também consegue determinar uma aplicação para poder alcançar de uma forma independente pelo número de processadores.

$$S_{(n,p)} = \frac{T_S(n)}{T_p(p,n)}$$

Figura 2: Equação da fórmula do cálculo *Speedup.* Fonte: Acervo do autor.

Na ilustração da figura 2, o *Speedup* é utilizado com a lei de Amdahl para que ele possa tomar em uma medida do grau de desempenho e ele consegue medir entre o tempo de execução serial e paralelo. A fórmula representa que o tempo serial é dividido com o tempo paralelo onde ele consegue totalizar o tempo de desempenho dos processadores.

3. Implementação

A implementação do método de ordenação do *Bubble Sort* na linguagem de programação em C/C++, foi implementado no ambiente do sistema operacional *Ubuntu* no arquivo do editor de texto *gedit*. E foi salvado as implementações na forma serial e paralelo no *gedit* com o arquivo de extensão da linguagem C/C++(.c).(github.com/KaelCordeiro?tab=repositories).

Listagem 2: Implementação do *Bubble Sort* na forma serial. Fonte: Acervo do autor.

```
0 #include <stdio.h>
1 #include <stdlib.h>
2 #include <time.h>
3
4 #define MAX_SIZE 100
5
6 void Bubble(int vet[]){
7 int j;
8 int i;
9 int aux;
10    for(i = 0; i < MAX_SIZE; i++){
11        for(j = 0; j < MAX_SIZE - 1; j++){
12             if(vet[j] > vet[j + 1]){
13                aux = vet[j];
14                vet[j] = vet[j + 1];
15                vet[j + 1] = aux;
16             }
17        }
18    }
19 }
20
21 int main(void){
22
23 int vet[MAX_SIZE];
24     srand(time(NULL))
25     for(i = 0; i < MAX_SIZE; i++){
26          vet[i] = rand() % 100;
27     }
28     clock_t inicio = clock();
29     Bubble(vet);
30     clock_t fim = clock();
31     double tempo = (double)(fim - inicio)/CLOCKS_PER_SEC;
32     printf("tempo total serial - %g", tempo);
```

```
33     printf("\n\n");
34
35     return 0;
36 }
```

Na listagem 2, a implementação do *Bubble Sort* em serial, foi adicionado o *#include <time.h>* que ele consegue pegar o tempo através de uma semente randômica onde o comando aparece na linha 24 que é o *srand(time(NULL))*. Para conseguir calcular a função do tempo em serial, foi utilizado o dado *clock_t* onde ele pega o inicio e o fim do tempo que foi jogado no código na linha 31 para ter uma equação para calcular o total do tempo em serial. Nesse cálculo, a subtração foi o fim do tempo e início do tempo, e depois ele pega o resultado da subtração e divide por segundos com a utilização do dado *CLOCKS_PER_SEC*.

Listagem 3: Implementação do *Bubble Sort* na forma paralela. Fonte: Acervo do autor.

```
0 #include <stdio.h>
1 #include <stdlib.h>
2 #include <time.h>
3 #include <omp.h>
4
5 #define MAX_SIZE 100
6 #define NUM_THREADS 2
7
8 void Bubble(int vet[]){
9 #pragma omp parallel num_threads(NUM_THREADS) shared(vet)
10 {
11 int j;
12 int i;
13 int aux;
14 #pragma omp for
15    for(i = 0; i < MAX_SIZE; i++){
16        for(j = 0; j < MAX_SIZE - 1; j++){
17             if(vet[j] > vet[j + 1]){
18                aux = vet[j];
19                vet[j] = vet[j + 1];
20                vet[j + 1] = aux;
21                }
22             }
23        }
24    }
25 }
26
27 int main(void){
28
29 int vet[MAX_SIZE];
30     srand(time(NULL))
31     for(i = 0; i < MAX_SIZE; i++){
32          vet[i] = rand() % 100;
33     }
34     double inicio = omp_get_wtime();
```

```
35      Bubble(vet);
36      double fim = omp_get_wtime();
37      double tempo = (double)(fim - inicio);
38      printf("tempo total paralelo - %g", tempo);
39      printf("\n\n");
40
41      return 0;
42  }
```

Na listagem 3, a implementação do *Bubble Sort* em paralelo, foi adicionado o *#include <omp.h>* para ele possa ser utilizado a biblioteca do *OpenMP* para o uso das diretivas, cláusulas e sincronizações. Nesse código, foi adicionado o *#pragma omp parallel* para que ele possa paralelizar o código fonte e foi usado a cláusula *num_threads* para que a quantidade das *threads* do *Bubble Sort* conseguem executar o código na região paralela com os parâmetros e foi compatível com a outra cláusula que é o *shared* para que os vetores do *Bubble Sort* consegue ter o endereço de memória acessível por todas as threads dentro da região paralela. No código fonte, na linha 14 foi adicionado o compartilhamento *#pragma omp for* para que ele consegue pegar os dois laços de iteração do *Bubble Sort* para conseguir situar a região paralela para que o laço se encontra e precisam ser executadas em paralelo. Para conseguir calcular a função do tempo em paralelo, foi utilizado o dado *omp_get_wtime()* onde possa obter o início e o fim do tempo que foi jogado no código nas linhas 34 e 36 para ter uma equação para calcular o total do tempo em paralelo. Nesse cálculo, apenas foi utilizado uma operação de subtração entre o fim do tempo e o início do tempo para conseguir obter o resultado do tempo paralelo.

4. Testes

4.1. Gráficos e Tabelas

Tamanho dos vetores	Tempo Serial
10	0,000006
20	0,000011
30	0,000018
40	0,00002
50	0,000047
60	0,000063
70	0,000081
80	0,000105
90	0,000176
100	0,000159

1000	0,012941
10000	0,978198
100000	103,735

Figura 3: Tabela de dados dos testes em serial do *Bubble Sort.* Fonte: Acervo do autor.

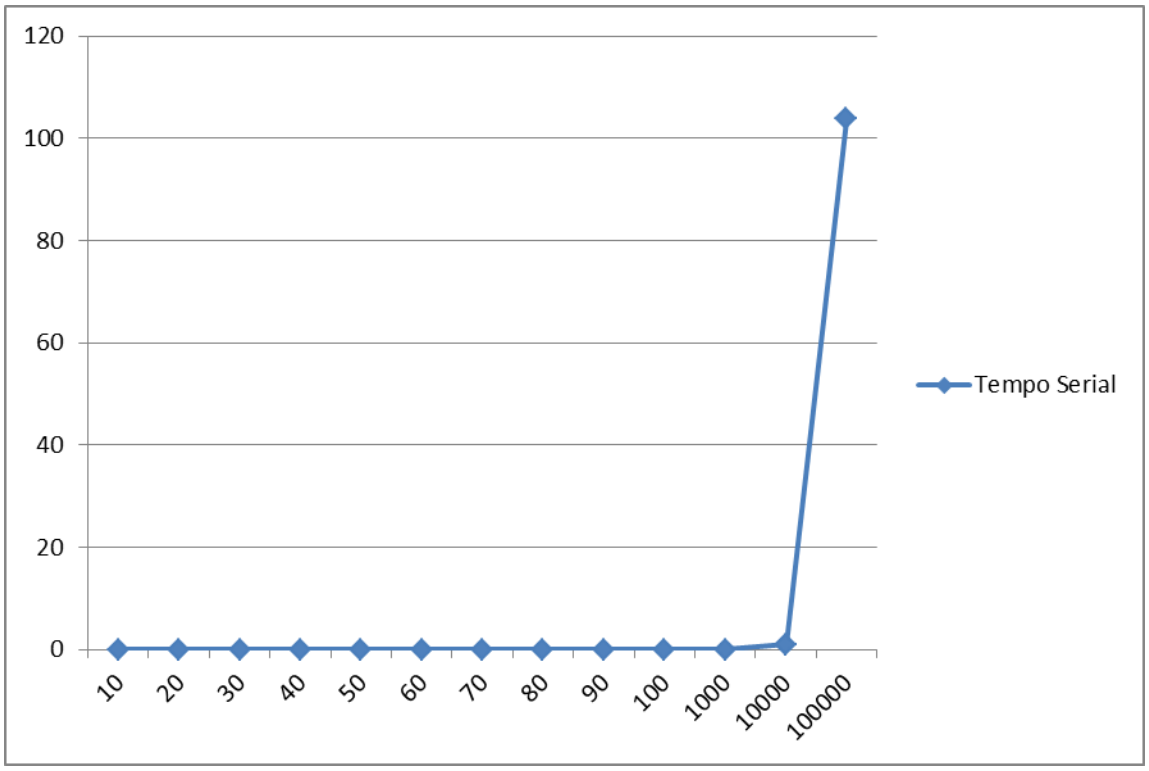

Figura 4: Gráfico dos testes em serial do *Bubble Sort.* Fonte: Acervo do autor.

Tamanho do vetor com as threads	Tempo Paralelo
10 com 1 thread	0,000018062
10 com 2 threads	0,000138641
10 com 4 threads	0,000219371
10 com 8 threads	0,00039594
10 com 16 threads	0,000711311

Figura 5: Dados da tabela com o tamanho 10 dos testes em paralelo do *Bubble Sort.* Fonte: Acervo do autor.

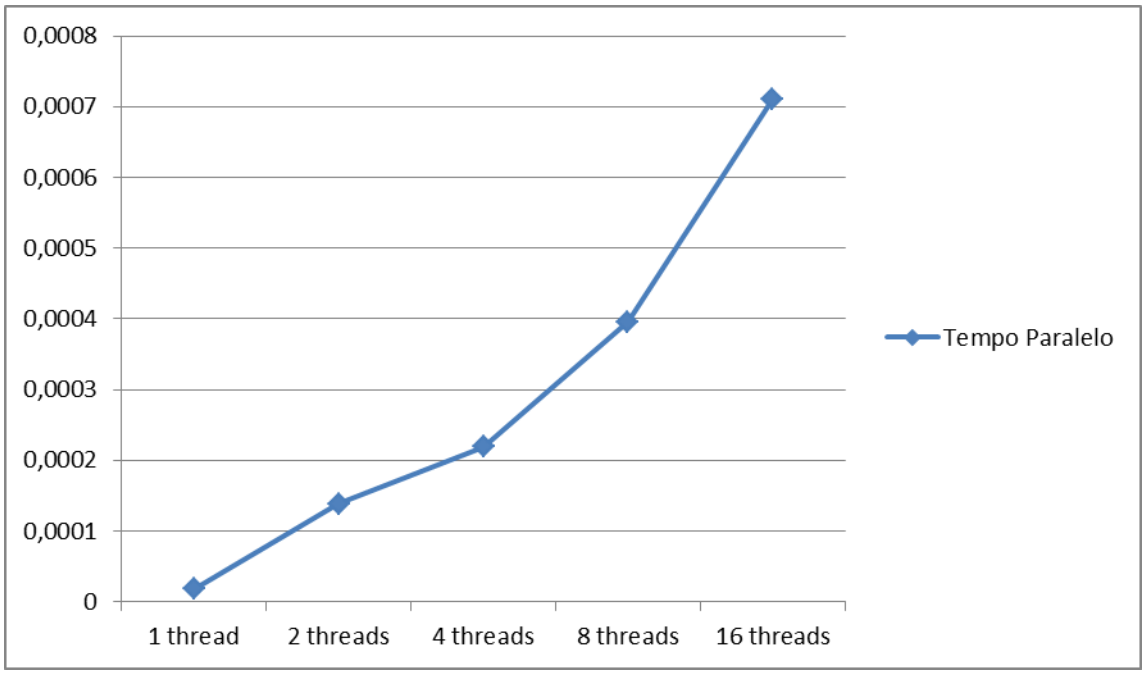

Figura 6: Gráfico dos testes com o tamanho 10 em paralelo do *Bubble Sort.*
Fonte: Acervo do autor.

Tamanho do vetor com as threads	Tempo Paralelo
100 com 1 thread	0,000187585
100 com 2 threads	0,00026097
100 com 4 threads	0,000455489
100 com 8 threads	0,000552928
100 com 16 threads	0,000867675

Figura 7: Dados da tabela com o tamanho 100 dos testes em paralelo do *Bubble Sort.*
Fonte: Acervo do autor.

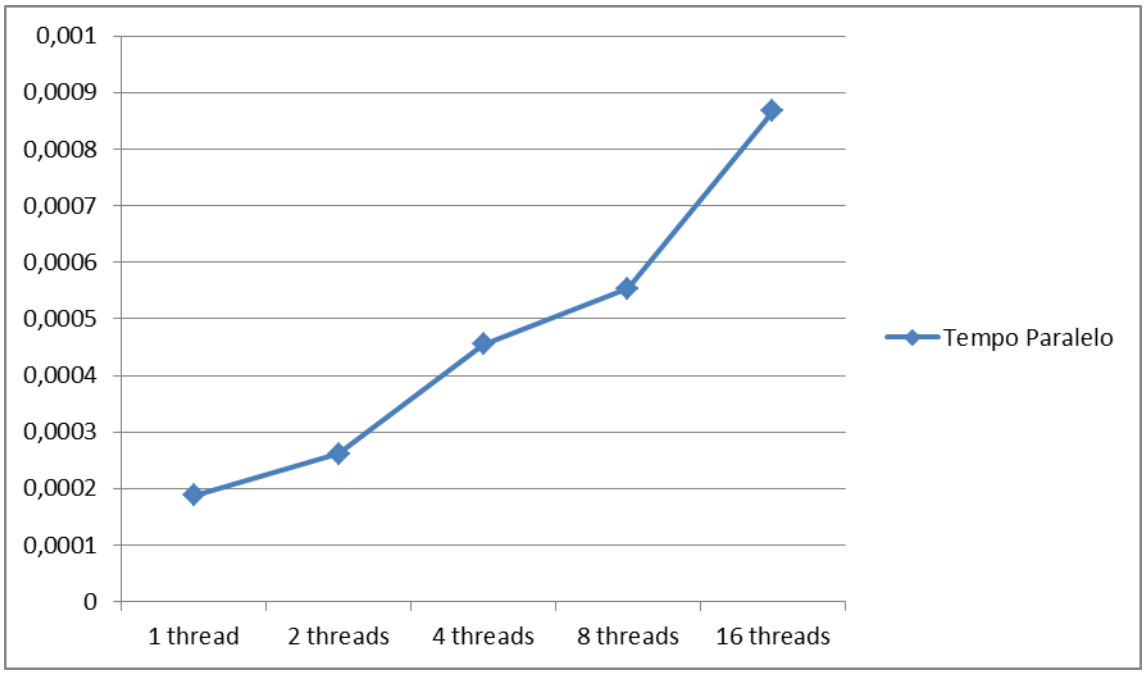

Figura 8: Gráfico dos testes com o tamanho 100 em paralelo do *Bubble Sort.*
Fonte: Acervo do autor.

Tamanho do vetor com as threads	Tempo Paralelo
1000 com 1 thread	0,0168063
1000 com 2 threads	0,0130792
1000 com 4 threads	0,0140631
1000 com 8 threads	0,014049
1000 com 16 threads	0,0143983

Figura 9: Dados da tabela com o tamanho 1000 dos testes em paralelo do *Bubble Sort.*
Fonte: Acervo do autor.

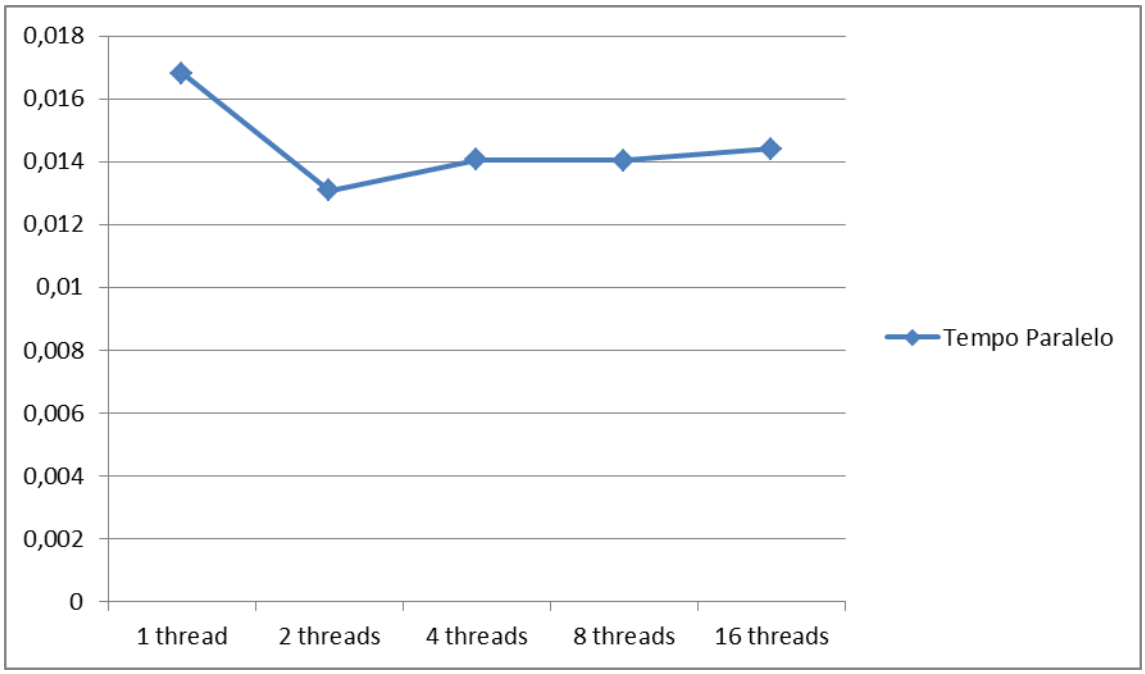

Figura 10: Gráfico dos testes com o tamanho 1000 em paralelo do *Bubble Sort.*
Fonte: Acervo do autor.

Tamanho do vetor com as threads	Tempo Paralelo
10000 com 1 thread	1,10198
10000 com 2 threads	0,551732
10000 com 4 threads	0,563397
10000 com 8 threads	0,57657
10000 com 16 threads	0,573203

Figura 11: Dados da tabela com o tamanho 10000 dos testes em paralelo do *Bubble Sort.*
Fonte: Acervo do autor.

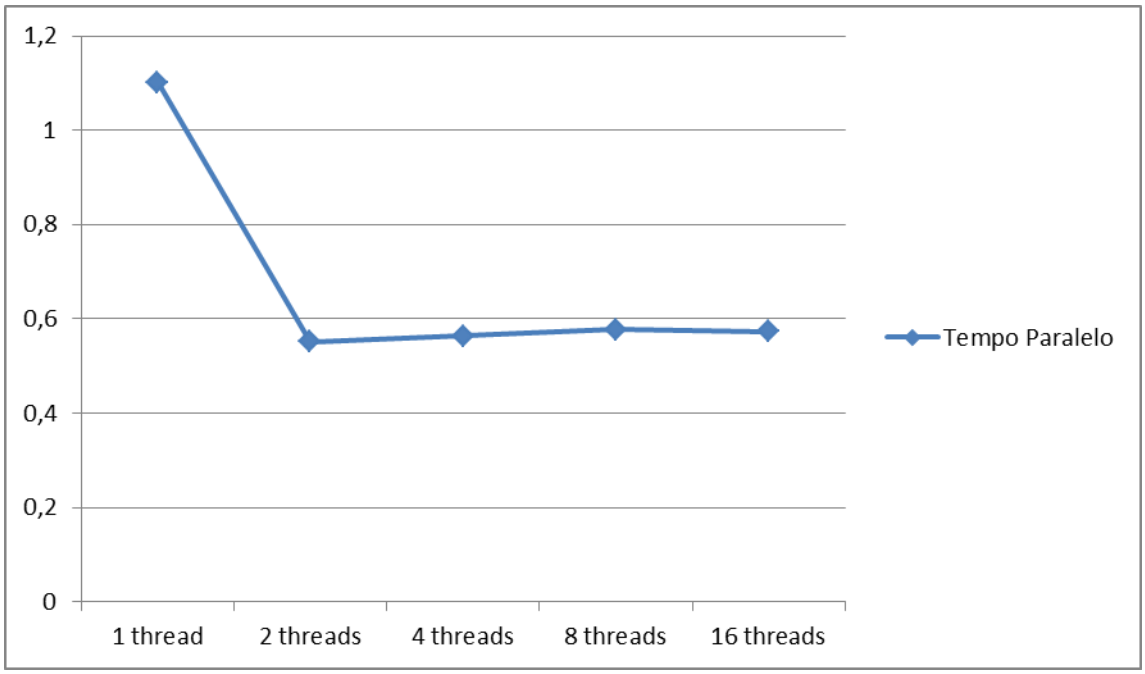

Figura 12: Gráfico dos testes com o tamanho 10000 em paralelo do *Bubble Sort.* Fonte: Acervo do autor.

Tamanho do vetor com as threads	Tempo Paralelo
100000 com 1 thread	110,681
100000 com 2 threads	55,5881
100000 com 4 threads	55,4231
100000 com 8 threads	55,2416
100000 com 16 threads	55,1797

Figura 13: Dados da tabela com o tamanho 100000 dos testes em paralelo do *Bubble Sort.* Fonte: Acervo do autor.

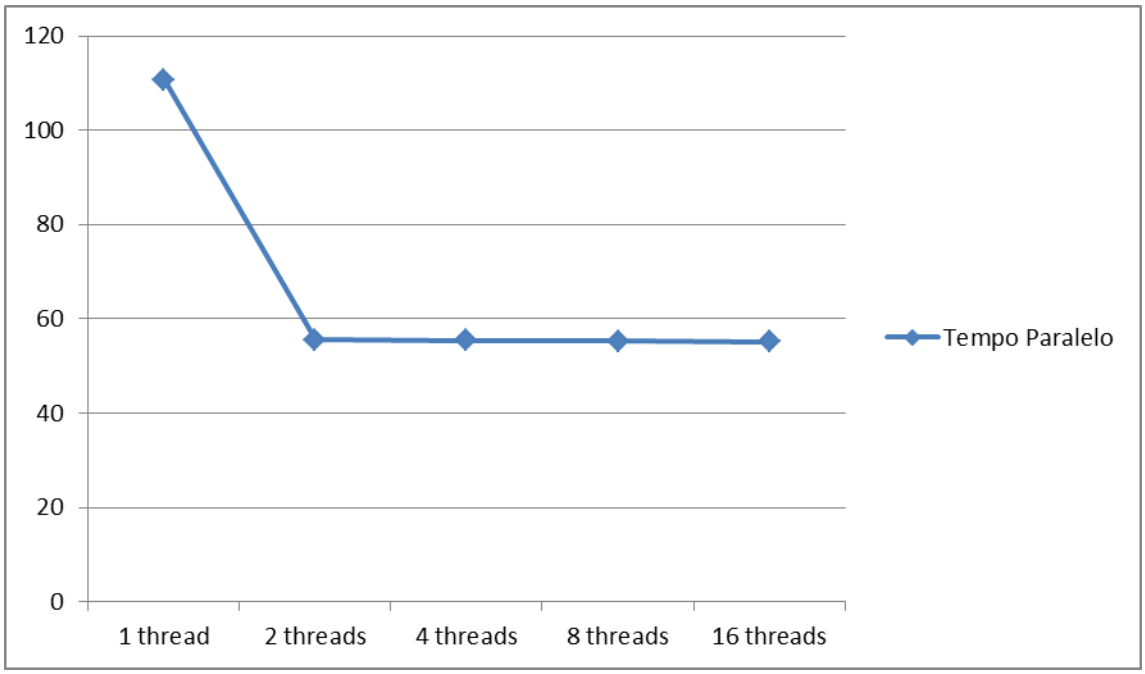

Figura 14: Gráfico dos testes com o tamanho 100000 em paralelo do *Bubble Sort.*
Fonte: Acervo do autor.

Tamanho do vetor 10	Tempo Speedup
1 thread	0,33
2 threads	0,04
4 threads	0,02
8 threads	0,02
16 threads	0,01

Figura 15: Dados da tabela com o tamanho 10 do *Speedup* do *Bubble Sort.*

Fonte: Acervo do autor.

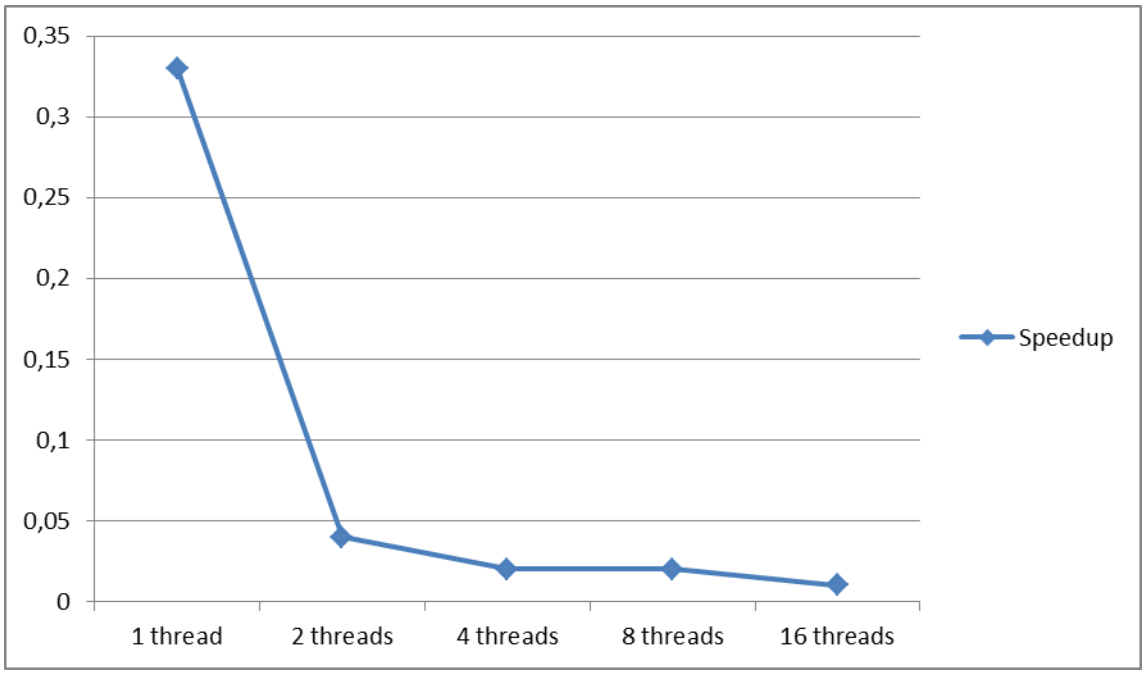

Figura 16: Gráfico do *Speedup* com o tamanho 10 do *Bubble Sort.*
Fonte: Acervo do autor.

Tamanho do vetor 100	Tempo Speedup
1 thread	0,85
2 threads	0,61
4 threads	0,35
8 threads	0,28
16 threads	0,18

Figura 17: Dados da tabela com o tamanho 100 do *Speedup* do *Bubble Sort.*

Fonte: Acervo do autor.

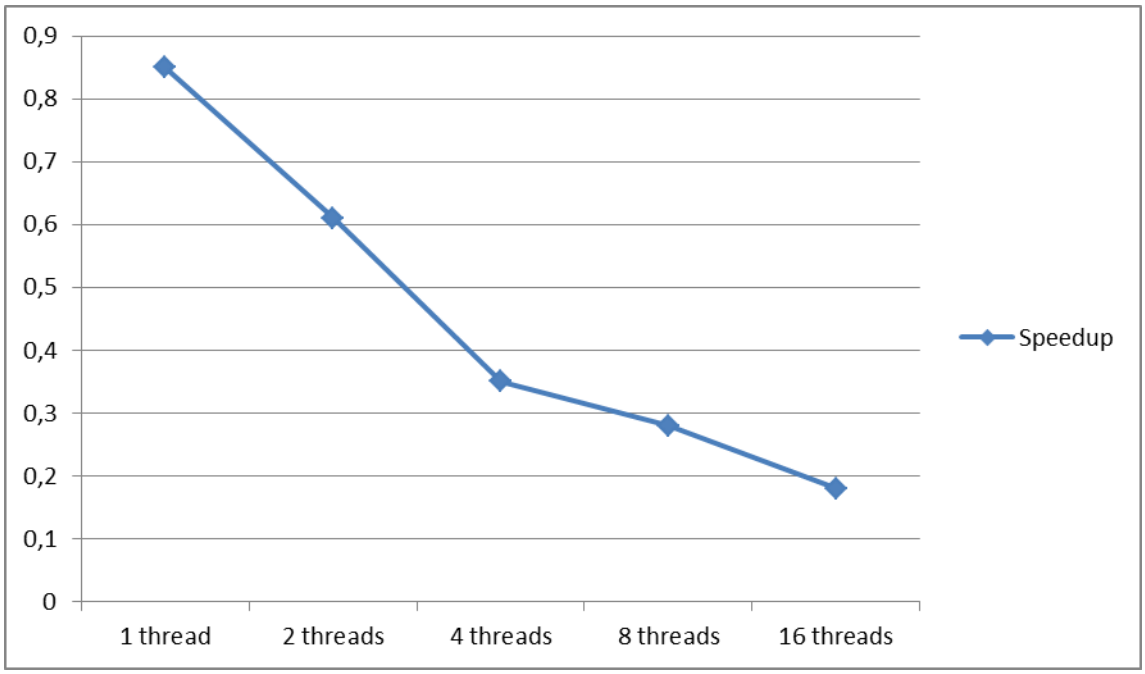

Figura 18: Gráfico do *Speedup* com o tamanho 100 do *Bubble Sort.*
Fonte: Acervo do autor.

Tamanho do vetor 1000	Tempo Speedup
1 thread	0,77
2 threads	0,99
4 threads	0,92
8 threads	0,92
16 threads	0,89

Figura 19: Dados da tabela com o tamanho 1000 do *Speedup* do *Bubble Sort.*

Fonte: Acervo do autor.

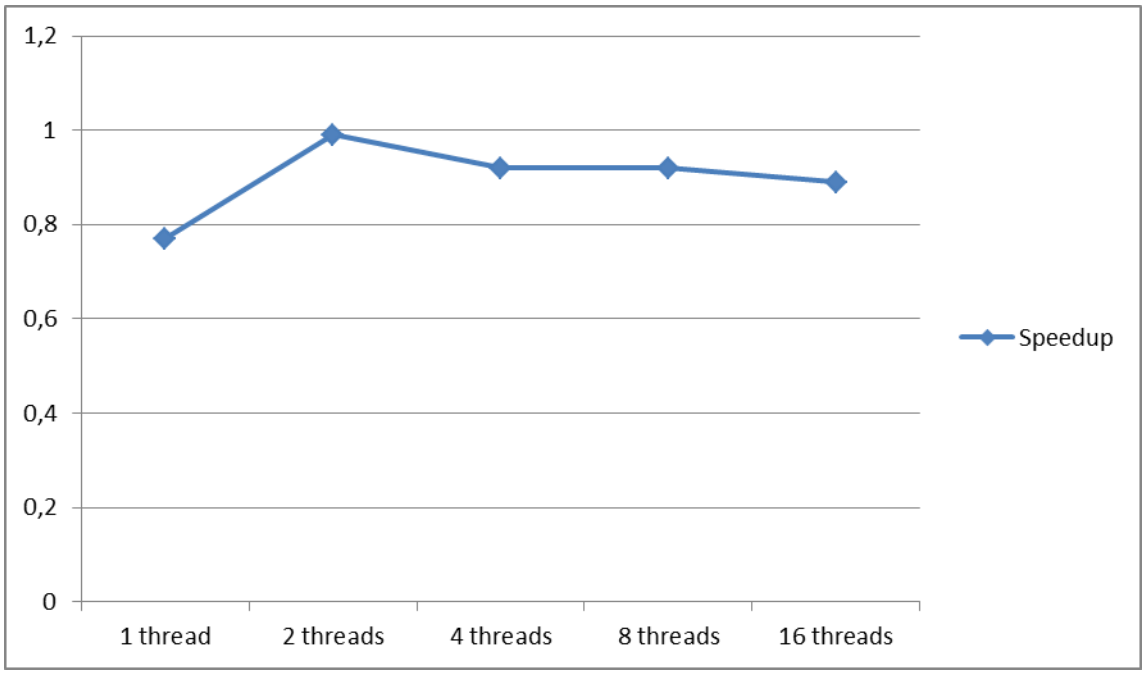

Figura 20: Gráfico do *Speedup* com o tamanho 1000 do *Bubble Sort.*
Fonte: Acervo do autor.

Tamanho do vetor 10000	Tempo Speedup
1 thread	0,88
2 threads	1,77
4 threads	1,74
8 threads	1,69
16 threads	1,71

Figura 21: Dados da tabela com o tamanho 10000 do *Speedup* do *Bubble Sort.*

Fonte: Acervo do autor.

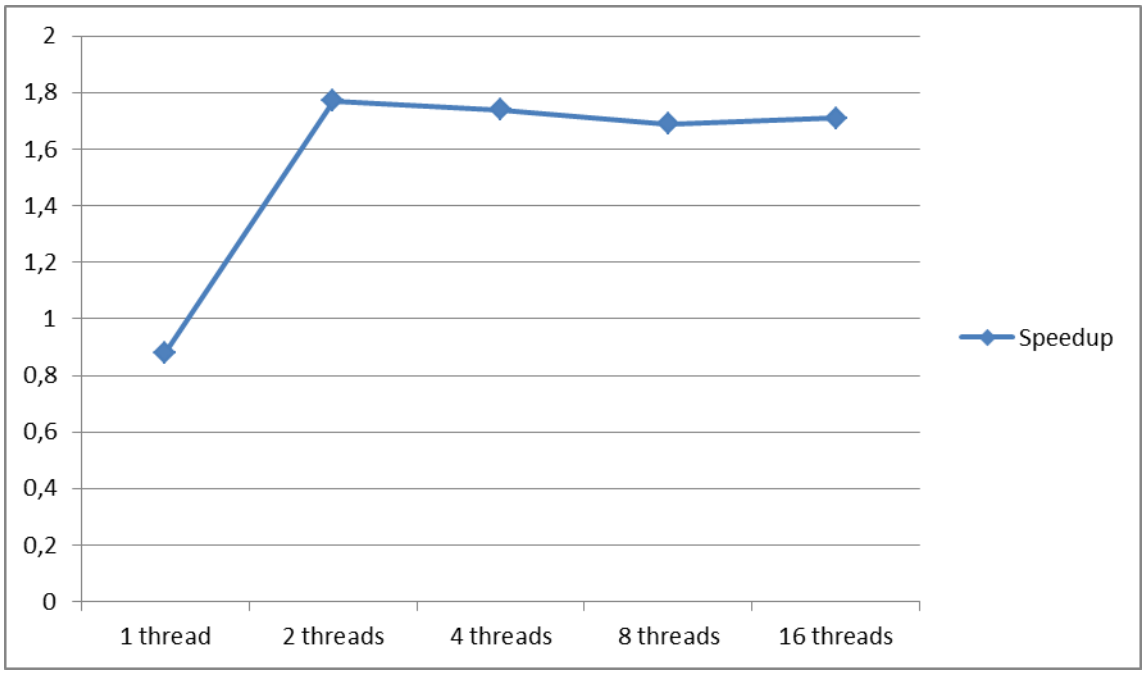

Figura 22: Gráfico do *Speedup* com o tamanho 10000 do *Bubble Sort.*
Fonte: Acervo do autor.

Tamanho do vetor 100000	Tempo Speedup
1 thread	0,94
2 threads	1,86
4 threads	1,87
8 threads	1,87
16 threads	1,88

Figura 23: Dados da tabela com o tamanho 100000 do *Speedup* do *Bubble Sort.*
Fonte: Acervo do autor.

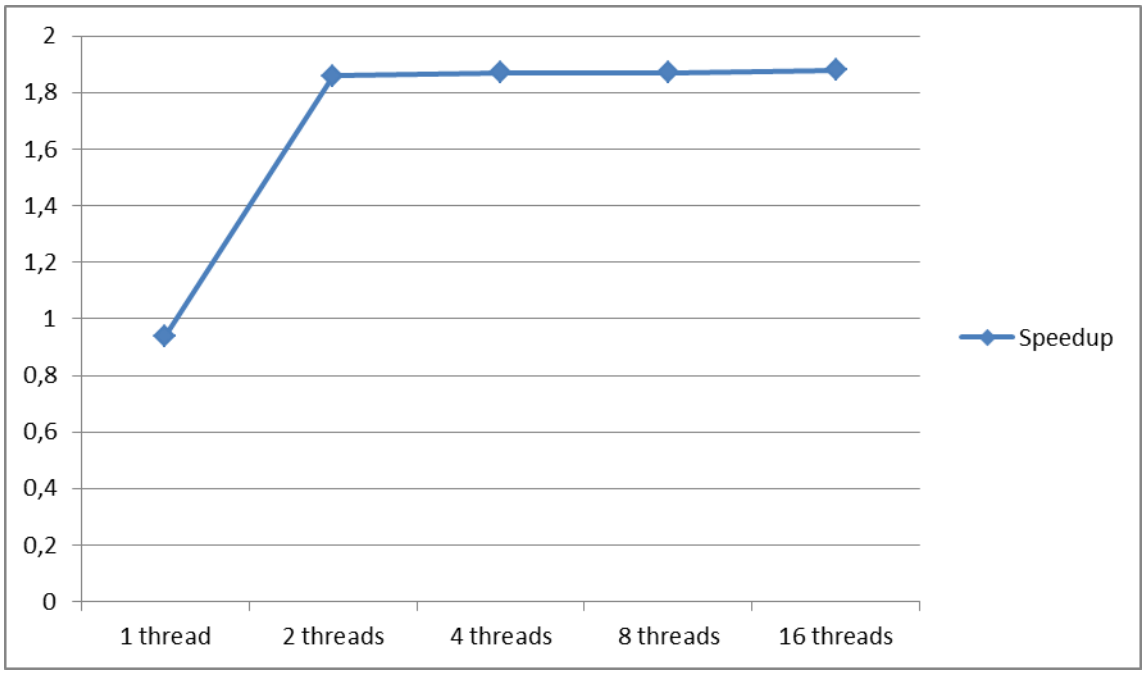

Figura 24: Gráfico do *Speedup* com o tamanho 10000 do *Bubble Sort.*
Fonte: Acervo do autor.

4.2. Resultados

Na figura 4, o gráfico mostra os resultados obtidos durante os testes do tempo de execução em serial e paralelo, os tamanhos dos vetores de 10 até 10000 o tempo deles foi mais rápido para serializar de cada tamanho dos vetores. Durante a execução do tempo com o tamanho do vetor 100000 na forma serial, ocorreu uma demora de um tempo em 3 minutos para serializar com o tamanho do vetor 100000. Os índices desse gráfico em cada tamanho dos vetores, e, o tamanho dos vetores 10 até 10000 aumentou pouco índice e já no tamanho do vetor 100000 houve um maior aumento do índice desse gráfico.

Os resultados do teste na versão paralela foi o tamanho dos vetores em cada *threads*. Porem, os vetores do tamanho 10, 100, 1000, 10000 passar mais rápido para coletar os dados durante esses testes.

Na figura 6, o gráfico mostra os resultados obtidos durante os testes do tempo de execução em paralelo com o tamanho do vetor 10 em cada *thread*, mas com as 16 *threads* com o tamanho desse vetor aumentou mais o índice desse gráfico.

Na figura 8, o gráfico mostra os resultados obtidos durante os testes do tempo de execução em paralelo com o tamanho do vetor 100 em cada *thread*, mas com as 16 *threads* com o tamanho desse vetor aumentou mais o índice desse gráfico.

Na figura 10, o gráfico mostra os resultados obtidos durante os testes do tempo de execução em paralelo com o tamanho do vetor 1000 em cada *thread*, já com 2 *threads* diminuiu o índice desse gráfico e com 4 *threads* ele aumentou o índice e já com

8 *threads* houve um pouco aumento e ainda mantem o índice desse gráfico, mas com as 16 *threads* com o tamanho desse vetor houve um pouco aumento desse gráfico.

Na figura 12, o gráfico mostra os resultados obtidos durante os testes do tempo de execução em paralelo com o tamanho do vetor 10000 em cada *thread*, já com 2 *threads* diminuiu o índice desse gráfico e com 4 *threads* ele aumentou o índice e já com 8 *threads* houve um pouco aumento do índice desse gráfico, mas com as 16 *threads* com o tamanho desse vetor houve em uma pouca diminuição desse gráfico.

Na figura 14, o gráfico mostra os resultados obtidos durante os testes do tempo de execução em paralelo com o tamanho do vetor 100000 em cada *thread*, já os resultados do tempo com 2, 4, 8 e 16 *threads* os índices desse gráfico caíram mais devido ao tempo de 2 e 3 minutos para paralelizar eles com o tamanho do vetor 100000. Enquanto isso, o resultado do teste em paralelo com o tamanho do vetor 100000 com as suas *threads*, não passaram bem rápido, devido uma demora de sair os resultados do tempo em 2 e 3 minutos.

Na figura 16, mostra o gráfico do *Speedup* com o tamanho do vetor 10 e os índices desse gráfico caiu bastante devido uma diminuição em segundos.

Na figura 18, mostra o gráfico do *Speedup* com o tamanho do vetor 100 e os índices desse gráfico caiu bastante devido uma diminuição em segundos que foi levado ao mesmo no tamanho do vetor 10, mas os números em segundos deles foram diferentes.

Na figura 20, mostra o gráfico do *Speedup* com o tamanho do vetor 1000 e os índices desse gráfico mudou em vários segundos, com 2 *threads* aumentou o índice desse gráfico, e, já com 4 *threads* e 8 *threads* ficaram com o mesmo resultado em segundos que foi de 0,92 e já com as 16 *threads* levou uma pouca diminuição do tempo em segundos que resultou em 0,89 segundos devido ao resultado anteriormente com as 4 e 8 *threads* que levou em 0,92 e caiu em 0,03 segundos para as 16 *threads*.

Na figura 22, mostra o gráfico do *Speedup* com o tamanho do vetor 10000 e os índices desse gráfico mudou em vários segundos, com 2 *threads* aumentou o índice desse gráfico, e, já com 4 *threads* e 8 *threads* diminuiu o índice. E com as 16 *threads*, o índice subiu para 1,71 segundos devido ao resultado anteriormente com as 8 *threads* que levou em 1,69 e conseguiu subir em 0,02 segundos para as 16 *threads*.

Na figura 24, mostra o gráfico do *Speedup* com o tamanho do vetor 100000 e os índices desse gráfico mudou em vários segundos, com 2 *threads* aumentou o índice desse gráfico, e, já com 4 *threads* e 8 *threads* ficaram com o mesmo resultado em segundos que foi de 1,87 e já com as 16 *threads* levou em um pouco aumento de tempo em segundos que resultou em 1,88 segundos devido ao resultado anteriormente com as 4 e 8 *threads* que levou em 1,87 e subiu em 0,01 segundos para as 16 *threads*.

5. Conclusão

O desenvolvimento deste artigo demonstra a complexidade do algoritmo, o que não o aconselha para grandes volumes de dados. De outro entendimento, deixa bastante claro que para se ter o domínio do conteúdo, urge a necessidade de um estudo que possa contemplar teoria e prática, que ao final irão se complementar.

Referências

Chandra, Rohit; Dagum, Leonardo; Kohr, Dave; Maydan, Dror; McDonald, Jeff; Menon, Ramesh. **Parallel Programming in OpenMP**. Acesso 11/06/2019 às 14:45.

Felipe, Henrique. ***Bubble Sort***. www.blogcyberini.com/2018/02/bubble-sort.html. Acesso 14/06/2019 às 16:03.

Gonçalves, Rui Carlos. **Paralelização de aplicações com OpenMP**. www.revista-programar.info/artigos/paralelizacao-de-aplicacoes-com-openmp/. Acesso 14/06/2019 às 14:14.

Iverson, Kenneth. **Classificação de bolhas: uma análise algorítmica arqueológica**. users.cs.duke.edu/~ola/bubble/bubble.html. Acesso 14/06/2019 às 14:55.

Rocha, Ricardo. **Métricas de Desempenho**. www.dcc.fc.up.pt/~ricroc/aulas/0708/ppd/apontamentos/metricas.pdf. Acesso 15/06/2019 às 13:47.

Szwarcfiter, Jayme Luiz. **Estruturas de dados e seus algoritmos** / Jayme Luiz Szwarcfiter, Lilian Markenzon. – 3.ed. – [Reimpr]. – Rio de Janeiro LTC, 2012. Acesso 14/06/2019 às 15:12.