Heapsort paralelizado com OpenMP

Felipe Tenfen[1], Rodrigo Curvello[2]

[1, 2]Instituto Federal de Educação, Ciência e Tecnologia Catarinense (IFC) – Câmpus Rio do Sul
Cep 89160-202 – Rio do Sul – SC – Brasil

felipetenfen@hotmail.com[1], rodrigo.curvello@ifc.edu.br

ABSTRACT. *The increasing development of high-performance processing, coupled with available computing resources and the use of parallel computing techniques, increasingly seeks to exploit the processing capacity of the chosen architecture to the maximum as well as to reduce processing time-out. In this way, this work proposes to analyze the efficiency that can be obtained through the programming of high performance, using the algorithm of ordering Heapsort, implemented in language C and parallelized through the OpenMP API. We perform the serial and parallel implementations of the algorithm in question and the efficiency comparisons presented through the calculation of speed-up obtained through the execution times of the algorithm.*
Keywords: *High Performance Programming; Heapsort; Parallelism; Thread.*

RESUMO. *O crescente desenvolvimento no processamento de alto desempenho, associado aos recursos computacionais disponíveis e ao emprego de técnicas de computação paralela, busca cada vez mais explorar ao máximo a capacidade de processamento da arquitetura escolhida assim como reduzir o tempo de espera do processamento. Desta forma este trabalho propõe analisar a eficiência que pode-se obter por meio da programação de alto desempenho, através da utilização do algoritmo de ordenação Heapsort, implementado na linguagem C e paralelizado por meio da API OpenMP. São realizadas as implementações serial e paralela do algoritmo em questão e as comparações de eficiência apresentadas através do cálculo de speed-up obtido por meio dos tempos de execução do algoritmo.*
Palavras-chave: *Programação Alto Desempenho; Heapsort; Paralelismo; Thread.*

1. Introdução

Ao longo dos anos, o desenvolvimento científico tem exigido das simulações numéricas resultados cada vez mais confiáveis e próximos da realidade, onde a procura por processamento de alto desempenho, em geral, tem sido atendida por equipamentos ou sistemas computacionais

de custo elevado. Diante da popularização das GPUs (Graphics Processing Units) e a aplicação de técnicas consolidadas de programação paralela, diversas áreas de pesquisa como a computação científica, o processamento e a análise de imagem, e muitas outras, podem conquistar avanços ainda mais significativos, sem a necessidade de grandes investimentos financeiros (GULO, 2015).

Atualmente pode-se observar um grande crescimento no número de aplicações que exigem cada vez mais uma grande quantidade e capacidade no processamento de dados e para solucionar esta demanda por poder computacional busca-se desenvolver, melhorar e evoluir o processamento de alto desempenho.

A computação de Alto Desempenho consiste em uma técnica da computação científica que tem como objetivo aumentar a velocidade de processamento ou até mesmo viabilizar as simulações numéricas de problemas físicos complexos e com elevado grau de discretização (SENA e COSTA, 2008), e nesta área está contida a técnica de processamento paralelo.

Desta forma, existe um crescente avanço na oferta de recursos de alto desempenho no mercado, disponibilizando cada vez mais recursos de alto desempenho, através de supercomputadores, redes de computadores, aglomerados de computadores e, até mesmo, computadores pessoais com mais de um processador, cujo desempenho ultrapassa, em muito, o desempenho de vários supercomputadores do passado.

A partir das informações acima, o presente trabalho propõe-se realizar e apresentar um estudo de caso com o objetivo de analisar e verificar o desempenho computacional que pode-se obter por meio da utilização da programação paralela. Será realizada a implementação de duas versões do algoritmo de ordenação Heapsort, uma versão sendo implementada utilizando-se a programação sequencial e a outra a programação paralela através da API OpenMP, e por meio disto, é realizada a análise de desempenho através do cálculo de *spped-up* do algoritmo supracitado.

2. Fundamentação Teórica

2.1. Alto Desempenho

Em função da demanda sempre crescente por poder computacional, as máquinas paralelas vem tornando-se cada vez mais populares, porém, os sistemas que oferecem a capacidade de processamento para satisfazer esta demanda, muitas vezes, ainda têm um custo muito elevado e/ou são difíceis de programar. O estudo de arquiteturas paralelas contribui para o entendimento e busca de alternativas para os dois problemas.

No caso do custo, um melhor entendimento das alternativas de construção de máquinas paralelas, e a compreensão de como estas decisões de projeto repercutirão no desempenho final da máquina, podem possibilitar a escolha de uma arquitetura de menor custo que ainda obtenha o desempenho desejado.

Como a programação de aplicações paralelas ainda exige o conhecimento de características específicas da máquina para a obtenção de desempenho, sólidos conhecimentos sobre a arquitetura de máquinas paralelas auxiliam em todo o ciclo de desenvolvimento de programas paralelos, desde sua modelagem até a fase de depuração e otimização (ROSE; NAVAUX, 2004).

Os supercomputadores, por exemplo, são sistemas fortemente acoplados mais que possuem um custo elevado e, portanto não é usual a sua implementação em termos de custo. Uma alternativa viável para essa demanda é a utilização de cluster, também conhecido como sistema fracamente acoplado.

Segundo Ferreira, Py e Carvalho (2001), clusters são aglomerados de estações (por exemplo, PCs) interligados por redes de alto desempenho, que transformam várias máquinas em um único ambiente de programação concorrente.

2.2. Arquiteturas Paralelas

Segundo Duncan (1990), uma arquitetura paralela fornece uma estrutura explícita e de alto nível para o desenvolvimento de soluções utilizando o processamento paralelo, através da existência de múltiplos processadores que cooperam para resolver problemas através de execução concorrente.

Existem muitas maneiras de se organizar computadores paralelos, mais para melhor entender as diferentes arquiteturas paralelas pode-se utilizar de técnicas de classificação, como por exemplo a classificação de Flynn. Esta classificação baseia-se basicamente no fato de um computador executar uma sequência de instruções sobre uma sequência de dados, diferencia-se o fluxo de instruções e o fluxo de dados. Dependendo de esses fluxos serem múltiplos ou não, e através da combinação das possibilidades, Flynn propôs quatro classes:

SISD - Single Instruction/Single Data stream: nesta classe um único fluxo de instruções atua sobre um único fluxo de dados, o que enquadra as máquinas que possuem a arquitetura de

von Neumann com apenas um processador, como os antigos computadores pessoais e estações de trabalho (ROSE e NAVAUX, 2004);

MISD – Multiple Instruction/Single Data: nesse caso, múltiplos fluxos de instruções atuariam sobre um único fluxo de dados, ou seja, múltiplas unidades de processamento executam suas diferentes instruções sobre o mesmo fluxo de dados. Na prática, diferentes instruções operam a mesma posição de memória ao mesmo tempo, executando instruções diferentes, porém, como isso, até os dias de hoje, não faz qualquer sentido, além de ser tecnicamente impraticável, nenhum sistema se enquadra nessa categoria (ROSE e NAVAUX, 2004);

SIMD - Single Instruction/Multiple Data: corresponde aos processadores matriciais/vetoriais, onde uma única instrução é executada ao mesmo tempo sobre múltiplos dados. Segundo Rose e Navaux (2004), é importante ressaltar que, para que o processamento das diferentes posições de memória possa ocorrer em paralelo, a unidade de memória não pode ser implementada como um único módulo de memória, o que permitiria só uma operação por vez.

MIMD - Multiple Instruction/Multiple Data: nessa classe múltiplas instruções são executadas sobre múltiplos dados. E para que o processamento das diferentes posições de memória possa ocorrer em paralelo, a unidade de memória não pode ser implementada como um único módulo de memória. Essa classe engloba a maioria dos computadores paralelos.

Um outro critério que pode ser utilizado para a classificação de máquinas paralelas é referente ao compartilhamento da memória. Rose e Navaux (2004), afirmam que quando se fala em memória compartilhada, existe um único espaço de endereçamento que será usado de forma implícita para comunicação entre processadores e quando a memória não é compartilhada, existem múltiplos espaços de endereçamento privados, um para cada processador.

Existe também a memória distribuída e memória centralizada, que de acordo com Rose e Navaux (2004), refere-se à localização física da memória. Se a memória é implementada com vários módulos, e cada módulo foi colocado próximo de um processador, então a memória é considerada distribuída, porém se a memória encontra-se à mesma distância de todos os processadores, independentemente de ter sido implementada com um ou vários módulos, então é uma memória centralizada.

Algo que também merece destaque em maquinas de múltiplos processadores, é em relação ao tipo como os processadores acessam às memórias do sistema, que podem ser classificados como de arquitetura **UMA** (*Uniform Memory Access*), onde a memória pode ser acessada uniformemente e esta é compartilhada por todos os processadores ou **NUMA** (*Non-Uniform Memory Access*), onde a memória não é acessada uniformemente e cada processador é composto por uma memória sua.

Segundo Rose e Navaux (2004), a principal diferença das arquiteturas UMA e NUMA estão no tempo de acesso a memória cache, ou seja, em UMA o tempo de acesso a memória por um processador é o mesmo para todas as regiões da memória, já em NUMA o tempo de acesso memoria por um processador difere conforme a região da memória que está sendo usada.

2.3. Modelos de Programação Paralela

Segundo Barreto, Ávila e Oliveira (2001), os modelos de programação paralela dividem-se em um modelo de programação implícita e três modelos de programação explícita: Passagem de Mensagens, Paralelismo de Dados e Variáveis Compartilhadas.

De acordo com os autores supracitados, o *Paralelismo Implícito* acontece quando o compilador e/ou o sistema de execução a responsabilidade de explorar automaticamente o paralelismo existente no programa. Normalmente, o compilador detecta os trechos do programa que poderão ser executados em paralelo, através da análise de dependências. São exemplos o Kap e o Forge.

O modelo de *Paralelismo de Dados* é aplicável para modos de execução SIMD ou SPMD. Executam a mesma instrução ou trecho de programa com conjuntos de dados diferentes em nodos de processamento diferentes. Exemplos de linguagens são o Fortran 90 e o HPF.

Já o modelo de *Passagem de Mensagens* é definido com um programa que emprega passagem de mensagens em múltiplos processos, cada qual com seu fluxo de controle, executados de forma assíncrona. Os processos possuem espaços de endereçamento separados e o usuário especifica de forma explícita a carga e os dados para cada um deles. Exemplos de bibliotecas são o PVM e o MPI. Por fim 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 (BARRETO, ÁVILA E OLIVEIRA, 2001).

2.4. OpenMP

Segundo Sena e Costa (2008), o OpenMP consiste em uma interface de programação (API) e um conjunto de diretivas que permite a criação de programas paralelos com compartilhamento de memória através da implementação automática e otimizada de um conjunto de *threads*.

OpenMP é uma API, portável, baseada no modelo de programação paralela de memória compartilhada para arquiteturas de múltiplos processadores. OpenMP está disponível para uso com os compiladores C/C++ e Fortran, podendo ser executado em ambientes Unix e Windows (Sistemas Multithreads). Definido e mantido por um grupo composto na maioria por empresas de hardware e software, denominado como OpenMP ARB (Architecture Review Board) (CENAPAD, 2014).

O OpenMP não é uma linguagem de programação, ele representa um padrão que define como os compiladores devem gerar códigos paralelos através da incorporação, nos programas sequenciais, de diretivas que indicam como o trabalho será dividido entre os processadores ou *cores* (SENA e COSTA, 2008).

O desenvolvimento de programas OpenMP dá-se através de *diretivas*[1]. Por exemplo, nas linguagens C e C++ as diretivas são identificadas pelo *#pragma omp,* enquanto que na linguagem Fortran, as diretivas são identificadas pela sentinela *!$omp*. Caso o compilador não tenha suporte às diretivas OpenMP, no momento da compilação do código é simplesmente ignorado as diretivas e é compilado de forma sequencial (SENA e COSTA, 2008).

De acordo com Sena e Costa (2008) o uso do OpenMP proporciona algumas vantagens, como:

- Na maioria dos casos, são feitas poucas alterações no código serial existente;
- Possui uma robusta estrutura para suporte à programação paralela;
- Fácil compreensão e uso das diretivas;
- Suporte a paralelismo aninhado;
- Possibilita o ajuste dinâmico do número de *threads.*

[1] Consiste em uma linha de código com significado “especial” para o compilador.

2.5. Speed-up

Ao resolvermos um problema num sistema paralelo, o principal interesse é saber o ganho que teremos sobre a implementação serial deste problema. Para isso a aplicação de métricas que permitem avaliar a performance é muito usual.

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

Segundo Sena e Costa (2008), a principal métrica, das citadas acima, 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 o *speed-up*, mais rápido se encontra o código paralelo. A equação que define essa métrica é mostrada abaixo.

$$S_n = \frac{T_s}{T_n} \quad (1)$$

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

Porém, destaca-se conforme Sena e Costa (2008), 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, isso é ilustrado pela Lei de Amdahl, apresentada a seguir.

2.6. Lei de Amdahl

Nem todos os trechos do código são paralelizáveis, dessa forma pode-se sempre identificar, dentro de um código, uma região que sempre será executada em serial e uma outra que pode ser paralelizada. O aumento do número de processadores apenas influenciará no tempo necessário para executar a região passível de paralelização (SENA e COSTA, 2008).

Sena e Costa (2008) afirmam que a ideia da Lei de Amdahl é que existe sempre um limite ao qual a capacidade de ganho pela paralelização estará sujeita. Isso se deve a fatores como entrada e saída, dependência entre os dados e outros fatores intrínsecos à aplicação e à técnica de programação paralela utilizada. E este limite pode ser calculado por meio da equação abaixo apresentada:

$$S_p = T\ \frac{1}{S+\frac{(1-S)}{n}}\ onde\ lim_{n\to\infty} S_p = \frac{1}{S} \quad \begin{matrix} S-Fração\ serial\ do\ código \\ n-Número\ de\ processsadores \end{matrix} \qquad (2)$$

2.7. Algoritmo de Ordenação Heapsort

2.7.1. História

O algoritmo HeapSort é um algoritmo de ordenação generalista e pertencente à família dos algoritmos de ordenação por seleção, sendo desenvolvido por Robert W. Floyd e J.W.J. Williams.

Robert W. Floyd, nascido em Nova York em 08 de junho de 1936, Robert W. Floyd concluiu o ensino médio aos 14 anos. Com 17 anos, concluiu o curso de bacharelado em artes liberais na Universidade de Chicago e em 1958 recebeu o título de Bacharel em física. Em 1950 tornou-se membro da equipe da Fundação de Pesquisa Armour (hoje Instituto de Pesquisa IIT) no Illinois Institute of Technology. A partir da década de 60, iniciou uma longa jornada de publicações de muitos trabalhos notáveis, dentre esses, o desenvolvimento do algoritmo de ordenação HeapSort (GUIMARÃES, 2013).

Uma versão inicial do HeapSort foi publicada por Robert W. Floyd em 1962 sob o nome Treesort. Dois anos mais tarde, J.W.J. Williams publicou uma versão melhorada sob o nome HeapSort. E em dezembro de 1964, Robert W. Floyd publicou uma versão final com o nome Treesort 3.

2.7.2. Funcionamento

O projeto por indução que leva ao Heapsort é essencialmente o mesmo do Selection Sort: seleciona-se e posiciona-se o maior (ou menor) elemento do conjunto e então é aplicado a hipótese de indução para ordenar os elementos restantes (SOUZA, 2009).

A diferença importante é que no Heapsort utilizamos a estrutura de dados heap para selecionar o maior (ou menor) elemento eficientemente, onde o heap trata-se de um vetor que simula uma árvore binária completa, a menos, talvez, do último nível, com estrutura de heap (BEDER, 2008).

Segundo Beder (2008): “A estrutura de dados Heap (binário) é um vetor que pode ser visto, como uma árvore binária quase completa”.

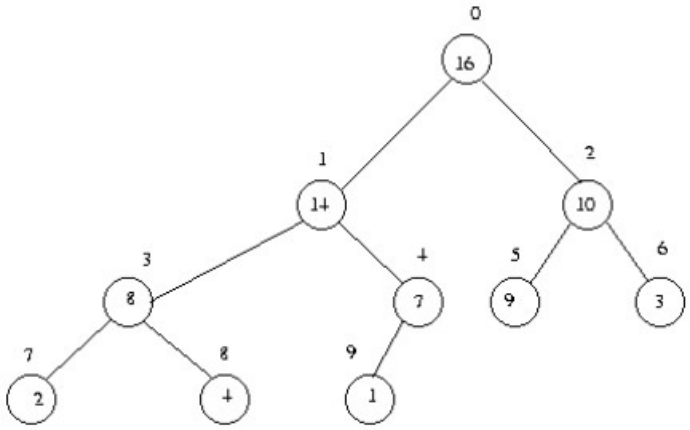

Figura 1 – Estrutura de dados Heap. Fonte: Beder (2008)

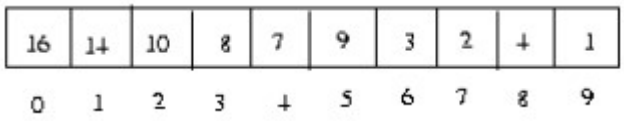

Figura 2 - Vetor de elementos da estrutura Heap. Fonte: Beder (2008)

De acordo com Beder (2008), cada nó da árvore corresponde a um elemento do vetor que armazena o valor no nó. A árvore está completamente preenchida em todos os níveis, exceto possivelmente no nível mais baixo, que é preenchido da esquerda para a direita até certo ponto.

Conforme Beder (2008) explica, para um dado vetor X, por exemplo, representar uma estrutura heap, é necessário verificar dois parâmetros:

- Comprimento de X (X.length): tamanho total do vetor;
- Comprimento do heap (heapComp): comprimento da parte do vetor que contém elementos da estrutura heap.

Segundo Mello (2002), há dois tipos de heap binário:

- *Heap máximo:* onde A[pai(i)] ≥ A[i], ou seja, o valor de um nó é no máximo o valor de seu pai, de forma que o maior do heap está na raiz e as sub árvores de um nó possuem valores menores ou iguais ao do nó.
- *Heap mínimo:* onde A[pai(i)] ≤ A[i], ou seja, o valor de um nó é no máximo o valor de seu pai, de forma que o menor elemento do heap está na raiz e as sub árvores de um nó possuem valores maiores ou iguais ao do nó.

Desta forma o algoritmo HeapSort basicamente utiliza uma estrutura de dados chamada de heap, que consiste em uma arvore cuja raiz é o elemento de maior valor, caso seja um heap máximo, ou o de menor valor, caso seja um heap mínimo (Mello, 2002).

O pseudoalgoritmo acima mostrado, tem como objetivo transformar um dado vetor em um vetor do tipo heap, a partir disso ele recebe como entrada um vetor, e sua respectiva posição inicial e final correspondente, em seguida, utiliza-se variável *aux* para armazenar a primeira posição do vetor, que no caso, é considerado o pai, e já na sequencia calculo o primeiro filho, que é armazenado na variável j. A partir disto entra-se no laço while, caso o filho se como pode ser observado na linha onde em sequência vou calcular

Com o heap construído o Heapsort apenas cria o heap a partir do vetor de entrada, e o algoritmo apenas vai removendo do heap e colocando no vetor de saída, onde ocorre o processo de ordenação, que de acordo com Mello (2008), a cada iteração seleciona o maior elemento do heap (sempre está na primeira posição) e o troca com o elemento no final do segmento não-ordenado. Após a troca, o novo elemento raiz do heap deve ser ajustado. O processo termina quando o heap tiver somente 1 (um) elemento (vetor ordenado).

2.7.3. Complexidade

O HeapSort é um dos algoritmos com melhor desempenho e é bastante utilizado em sistemas operacionais no gerenciamento de memória e na promoção de filas de prioridade, pois possui desempenho em tempo de execução satisfatório em sequências aleatoriamente desordenadas, utiliza pouca memória e o seu desempenho em pior cenário é praticamente igual ao desempenho em cenário médio, isso porque possui complexidade O(n lg n).

Desta forma, torna-se uma ótima possibilidade de utilização, visto que muitos dos algoritmos de ordenação possuem desempenho insatisfatório no pior cenário, quer em tempo de execução, quer no uso de memória, como por exemplo, os algoritmos Bubble Sort e Selection Sort que possuem ordem de complexidade $O(n^2)$.

3. Implementação

Conforme já mencionado anteriormente, foi realizada a implantação do algoritmo de ordenação Heapsort em duas versões, sequencial e paralela. Neste subitem serão apresentados alguns fragmentos de códigos utilizados na implementação do algoritmo, porém caso haja interesse em analisar melhor os códigos programados, e a forma como os dados foram

coletados, todo o código desenvolvido encontra-se hospedado e disponível de forma publica no repositório do Github, podendo ser acessado através do link: https://github.com/felipetenfen/Heapsort.

```
1 //Transforma um dado vetor em um vetor heap
2 void cria_heap(int *vet, int i, int f)
3 {
4   //Armazena a primeira posição do vetor, que é considerado o nó pai
5   int aux = vet[i];
6   //Calcula o primeiro filho e armazena na variavel j
7   int j = i * 2 + 1;
8
9   //Verifica se o fiho é menor ou igual a ultima posição do vetor
10  while (j <= f)
11  {
12      if (j < f)
13      {
14          //Compara qual o maior filho, do pai armazenado na
15          //variavel aux, e armazena na variavel j o maior
16          if (vet[j] < vet[j+1])
17          {
18              j = j + 1;
19          }
20      }
21
22      //Ve se o filho na posição vet[j], é maior que o pai, caso for,
23      //tranforma o filho no pai, e calcula seu próximo filho,
24      //senão encerra o j para sair do laço While
25      if ( aux < vet[j] )
26      {
27          vet[i] = vet[j];
28          i = j;
29          j = 2 * i + 1;
30      }
31      else
32      {
33          j = f + 1;
34      }
35  }
36
37  //No fim o antigo pai, ocupa o lugar do último filho analisado
38  vet[i] = aux;
39 }
```

Figura 3 – Método constroi Heap. Fonte: Elaborado pelo autor, 2019.

O método apresentado acima consiste basicamente em transformar um vetor qualquer em um vetor do tipo heap, que simula uma árvore binária completa, a menos, talvez, do último nível e cuja a raiz é o elemento de maior valor. O código encontra-se todo comentado e pode ser facilmente entendido, apenas observando-o.

Abaixo na Figura 4, é apresentado o método Heapsort paralelizado, que é responsável por realizar o processo de ordenação do vetor, após transforma-lo em uma estrutura heap. Todo o código encontra-se comentado, facilitando o entendimento de cada processo envolvido no método de ordenação. Na linha 4 é realizado o processo de paralelização do bloco de código, informando também o número de threads que serão utilizadas e as variáveis compartilhadas, e nas linhas 10 e 20, é feita a paralelização dos laços *for* existentes no método.

```
1 //Realiza o processo de ordenação de maneira paralelizada
2 void heap_sort(int *vet, int N)
3 {
4   #pragma omp parallel num_threads(NUM_THREADS) shared(vet, N)
5   {
6       int i, aux;
7
8       //Primeiro cria-se o vetor heap a partir dos dados, pegando
9       //o vetor do meio pra frente e tranformando-o em um heap
10      #pragma omp for
11      for ( i = (N -1)/2; i >= 0; i--)
12      {
13          cria_heap(vet, i, N-1);
14      }
15
16      //Pega o maior maior elemento da heap, que se encontra no topo da arvore,
17      //e coloca na ultima posição do vetor em seguida reestrutura toda a heap,
18      //sem considerar a ultima posição do vetor e realiza todo o processo
19      //novamente até que o vetor esteja ordenado
20      #pragma omp for
21      for ( i = N-1; i >= 1; i--)
22      {
23          aux = vet[0];
24          vet[0] = vet[i];
25          vet[i] = aux;
26
27          cria_heap(vet, 0, i-1);
28      }
29  }
30 }
```

Figura 4 – Método Heapsort. Fonte: Elaborado pelo autor, 2019.

4. Testes

Todos os testes efetuados neste estudo de caso foram realizados em um computador pessoal, no caso um notebook da marca Asus, modelo S46CB[2], equipado com um processador Intel® Core™ i7-3537U CPU 2.0Ghz, chipset Intel® HM76 Express, placa de vídeo dedicada NVIDIA® Série GeForce® GT 740M, 6GB de memória RAM DDR3 operando sobre o sistema operacional Windows 10 Home Single Language e utilizando o compilador GCC versão 6.3.0-1 para compilar as implementações realizadas.

Todos os dados que serão apresentados a seguir, foram obtidos a partir dos testes realizados através da implementação do algoritmo de ordenação Heapsort, na versão serial e paralela. Utilizou-se dois tipos de vetores de números inteiros, vetor aleatório e vetor ordenado inverso, com tamanhos definidos em: 10.000, 100.000, 1.000.0000 e 10.000.000. Foram

[2] Mais informações a respeito das configurações do equipamento podem ser obtidas em: https://www.asus.com/br/Laptops/S46CB/overview/

efetuadas 100 repetições para a versão serial e 100 repetições na versão paralela para cada grupo de threads de tamanhos 2, 4, 8, 16, 32, 64, 128 e 256.

Speed-up - Vetor Aleatório				
	Tamanho			
Threads	***10.000***	***100.000***	***1.000.000***	***10.000.000***
2	0,894281678	1,317401961	1,58056436	1,755655484
4	1,157894737	1,55513121	2,01192196	2,469443011
8	1,142871985	1,515642844	2,120305724	2,813894825
16	1,189197224	1,547341451	2,287261839	2,990981728
32	1,093167702	1,572085609	2,330406416	3,13880015
64	0,842101234	1,699483831	2,396013451	3,518541942
128	0,528530116	1,640808041	2,536075268	4,159595148
256	0,300340784	1,393515938	2,528563831	3,145099175

Tabela 1 - Speed-up dos vetores aleatórios de números inteiros. Fonte: Elaborado pelo autor, 2019.

Analisando a tabela 1, pode-se verificar que nos vetores menores de comprimento 10.000 e 100.000, indiferentemente do número de threads utilizadas, as taxas de *speed-up* obtidas não resultaram em um ganho de desempenho considerável em relação a execução sequencial.

Porém nos vetores maiores de comprimento 1.000.000 e 10.000.000, a partir da utilização de 4 ou mais *threads*, passou-se a obter melhores taxas de *speed-up,* e o resultado apresentado já podendo ser considerado como positivo, mostrando que neste caso a paralelização do algoritmo a partir deste ponto proporcionou um ganho de desempenho considerável e satisfatório na execução paralela em relação a execução sequencial

O caso que obteve a melhor taxa de *speed-up*, por sua vez, foi no vetor de comprimento 10.000.000, utilizando-se de 128 unidades de processamento (threads), onde obteve-se uma taxa de *speed-up* de 4,1595, ou seja, o algoritmo paralelizado neste caso, se apresentou 4,1595x mais rápido em relação a execução do algoritmo sequencial para um vetor de mesmo tamanho.

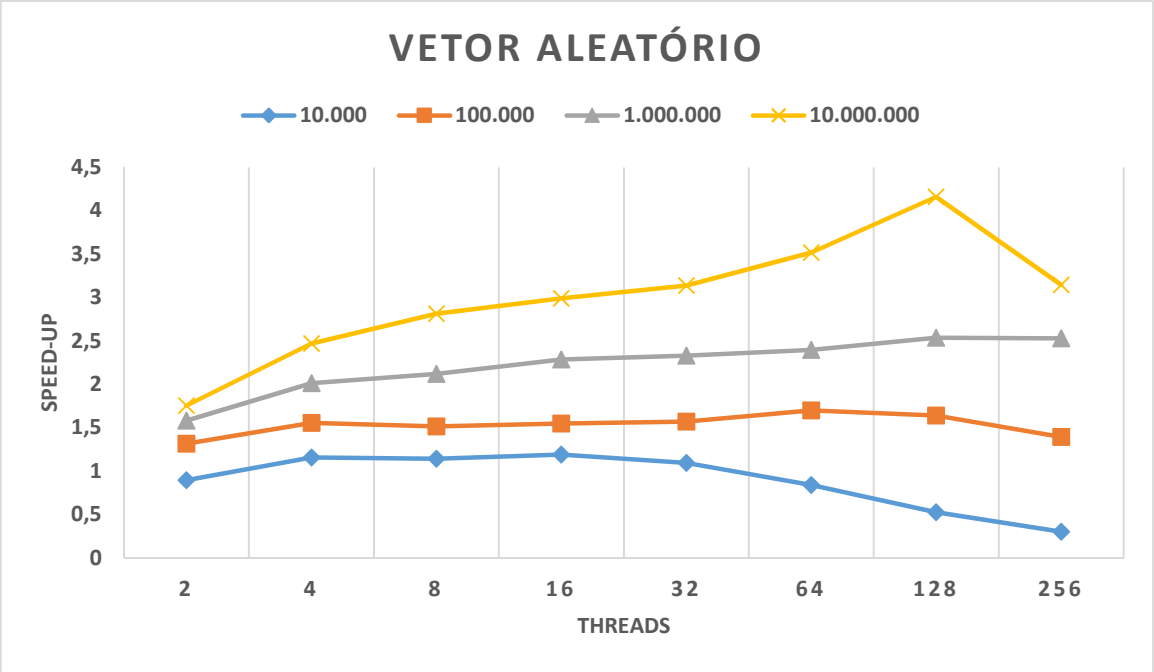

Gráfico 1 – Gráfico de linhas referente aos valores apresentados na Tabela 1. Fonte: Elaborado pelo autor, 2019.

Afim de facilitar a visualização dos resultados obtidos e apresentados na Tabela 1, acima é apresentado o gráfico de linha referente aos valores da tabela, aonde fica mais claro e perceptível, os resultados supracitados referentes ao algoritmo de ordenação Heapsort, utilizando como valores de entrada para ordenação vetores aleatórios de números inteiros.

A seguir na Tabela 2, são apresentadas as taxas de *speed-up* obtidas através da utilização de vetores ordenados inversos como valores de entrada para ordenação, onde neste caso por sua vez, os resultados no geral, tanto para os vetores de entrada menores quanto para os maiores, assim como para utilização de 2, 4, 8 ou até 256 threads, não apresentaram uma variação na taxa de *speed-up* considerável e que apresentação um ganho de desempenho satisfatório, comparando-se a versão paralela com a sequencial.

Speed-up - Vetor Ordenado Inverso				
Tamanho				
Threads	***10.000***	***100.000***	***1.000.000***	***10.000.000***
2	0,806060606	1,206059242	1,133629923	1,025339265
4	0,886666667	1,345532831	1,266950994	1,191588039
8	0,764363423	1,249383637	1,272377571	1,242985406
16	0,904761905	1,25546546	1,30333639	1,284675629
32	0,689122742	1,464923863	1,319848371	1,376418737
64	0,607303163	1,365433771	1,485357997	1,308424813
128	0,361413043	0,125781523	1,567640974	1,249771987
256	0,186797753	0,994104299	1,249645813	1,223922858

Tabela 2 - Speed-up dos vetores ordenados inversos de números inteiros. Fonte: Elaborado pelo autor, 2019.

Conforme mostrado abaixo no Gráfico 2, referente aos valores da Tabela 2, fica ainda mais perceptível que é pequena a variância entre as taxas de *speed-up* obtidas indiferentemente do tamanho do vetor utilizado ou do número de threads, exceto no vetor de tamanho 100.000, utilizando-se 128 threads, onde apresentou uma queda considerável da taxa de *speed-up,* obtendo um desempenho ruim.

Por fim, no geral os resultados mostraram que a paralelização do algoritmo de ordenação nesse caso, não representa ao ganho considerável de desempenho em relação a versão sequencial, ainda mais se levar em consideração os resultados obtidos nos vetores aleatórios como parâmetro de entrada para ordenação.

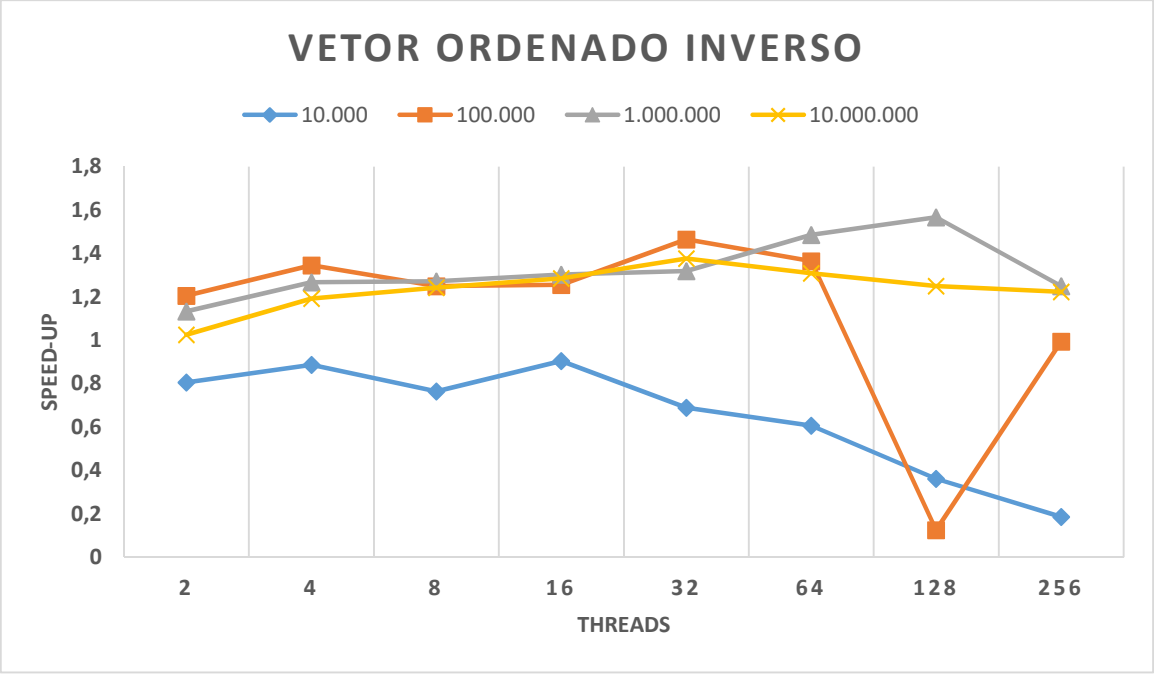

Gráfico 2 - Gráfico de linhas referente aos valores apresentados na Tabela 1. Fonte: Elaborado pelo autor, 2019.

5. Conclusão

Cada vez mais se exige computadores com arquiteturas avançadas e que ofereçam um processamento de alto desempenho, principalmente em áreas da engenharia e ciência, que exigem alta capacidade de processamento gráfico e matemático. Diante disto, o emprego do paralelismo mostrou-se essencial para se extrair todo o potencial dessas máquinas e, para isso, o domínio de técnicas e de linguagens de programação paralela são essenciais.

O aumento na disponibilidade de unidades de processamento, por sua vez, também tem sido muito importante para contribuir com aplicações que demandam alto poder de processamento.

Através do estudo de caso realizado nesse trabalho, pode-se verificar que é possível obter benefícios e vantagens na paralelização do algoritmo de ordenação Heapsort. Os cálculos de desempenho, mostraram que no caso dos algoritmos aleatórios de maior comprimento o emprego da paralelização proporciona um ganho de desempenho considerável e satisfatório, já nos algoritmos de menor comprimento o ganho foi menor, porém ainda assim apresentou ganho de desempenho se comparado a versão sequencial.

Nos vetores ordenados inverso, por sua vez, o ganho de desempenho se apresentou pequeno para os vetores de comprimento 100.000, 1.000.000 e 10.000.000 indiferentemente da quantidade de threads utilizadas, porém ainda assim apresentou um desempenho superior a versão sequencial. Já para o vetor de comprimento 10.000 todos os resultados obtidos foram abaixo da versão sequencial.

Por fim, diante de todas as informações apresentadas, mostrou que possuir o domínio de técnicas e de linguagens de programação paralela podem proporcionar ganhos de desempenho no processamento de dados, quando utilizados de maneira adequada, e pode-se dizer que todos esses motivos levam a crer que é cada vez mais significativa a importância do processamento paralelo e na Ciência da Computação, e sua presença é quase que imprescindível no mundo atual, onde exige-se processamento de alto desempenho.

6. Referências Bibliográficas

GULO, Carlos Alex S. J.. **Técnicas de Computação de Alto Desempenho para o Processamento e Análise Eficiente de Imagens Complexas.** 2015. Disponível em: <https://web.fe.up.pt/~tavares/downloads/publications/relatorios/Projeto-Tese-

CarlosGulo.pdf>. Acesso em: 08 abr. 2019.

SENA, M. C. R; COSTA, J. A. C. **Tutorial OpenMP C/C++.** Ed 01. Maceió: Sun Microsystens, 2008.

GUIMARÃES, Gleyser. **História da Computação:** O Algoritmo HeapSort. 2013. Disponível em: <http://www.dsc.ufcg.edu.br/~pet/jornal/maio2013/materias/historia_da_computacao.html>. Acesso em: 21 maio 2019.

BEDER, Delano M.. **Algoritmos de Ordenação:** HeapSort. 2008. Disponível em: <https://docplayer.com.br/66401635-Algoritmos-de-ordenacao-heapsort.html>. Acesso em: 21 maio 2019.

MELLO, Ronaldo S.. **Ordenação de Dados (III).** 2002. Disponível em: <http://www.inf.ufsc.br/~r.mello/ine5384/17-OrdenacaoDados3.pdf>. Acesso em: 21 maio 2019.

FEOFILOFF, Paulo. **HeapSort.** 2018. Disponível em: <https://www.ime.usp.br/~pf/algoritmos/aulas/hpsrt.html>. Acesso em: 21 maio 2019.

SOUZA, Jairo Francisco de. **HeapSort.** 2009. Disponível em: <http://www.ufjf.br/jairo_souza/files/2009/12/2-Ordena%C3%A7%C3%A3o-HeapSort.pdf>. Acesso em: 21 maio 2019.

DUNCAN, R., A **Survey of Parallel Computer Architectures,** IEEE Computer, pp.5-16, Fevereiro, 1990.

FERREIRA, Simone; PY, Mônica; CARVALHO, Elias. **Considerações Sobre o Ensino de Alto Desempenho.** 2001. Disponível em: <http://www.lbd.dcc.ufmg.br/colecoes/erad-rs/2001/009.pdf>. Acesso em: 08 abr. 2019.

ROSE, César A. F. de; NAVAUX, Philippe O. A.. **Fundamentos de Processamento de Alto Desempenho.** 2004. Disponível em: <http://www.lbd.dcc.ufmg.br/colecoes/erad-rs/2004/003.pdf>. Acesso em: 08 abr. 2019.

BRUSCHI, Sarita Mazzini. **Arquitetura de Computadores:** Arquiteturas Paralelas. Disponível em: <https://edisciplinas.usp.br/pluginfile.php/3482095/mod_resource/content/1/18aula%20-%20Arquiteturas%20Paralelas.pdf>. Acesso em: 03 jun. 2019.

BARRETO, Marcos E.; ÁVILA, Rafael B.; OLIVEIRA, Fábio A. D. de. **Execução de Aplicações em Ambientes Concorrentes.** Instituto de informática - Universidade do Rio Grande do Sul - Gramado - ERAD (2001). Acesso em: 03 jun. 2019.

CENAPAD. **Apostila de Treinamento:** Introdução ao OpenMP. 2014. Disponível em: <https://www.cenapad.unicamp.br/servicos/treinamentos/apostilas/apostila_openmp.pdf>. Acesso em: 03 jun. 2019.

ROCHA, Ricardo. **Programação Paralela e Distribuída:** Métricas de Desempenho. 2008. Disponível em: <https://www.dcc.fc.up.pt/~ricroc/aulas/0708/ppd/apontamentos/metricas.pdf>. Acesso em: 03 jun. 2019.