AVALIANDO A API OPENMP EM APLICAÇÕES PARALELAS DE MEMÓRIA COMPARTILHADA

Júlio Cesar Thomé Borges[1], Marcel Campos Inocencio[2]

Resumo: Com a crescente necessidade por poder de processamento é necessária uma opção alternativa ao aumento de clock, já que eventualmente chegaremos ao limite da tecnologia. A estratégia adotada, inicialmente em servidores, e atualmente disponível para computadores domésticos, foi a paralelização do problema, onde o mesmo é dividido em várias partes e executado em diferentes processadores ao mesmo tempo. Essa pesquisa avaliou a API OpenMP como uma solução para a dificuldade de desenvolvimento de algoritmos paralelos na linguagem C++, onde a mesma demonstrou que com poucas modificações no código e sem alterar a lógica do mesmo é possível paralelizar um algoritmo obtendo ganhos expressivos.

Palavras-chave: OpenMP. Computação Paralela. Multiprocessadores.

ABSTRACT: With the growing need for processing power, an alternative option to the clock increase was necessary, as we would eventually reach the limit of technology. The strategy adopted, initially in servers, and currently available for home computers, was the parallelization of the problem, where it is divided into several parts and executed in different processors at the same time. This research evaluated the OpenMP API as a solution to the difficulty of developing parallel algorithms in the C++ language, where it demonstrated that with few modifications to the code and without changing its logic, it is possible to parallelize an algorithm with significant gains.

Keywords: OpenMP. Parallel Computing. Multiprocessors.

[1] julio_borges@outlook.com.br.
[2] marcel.inocencio@gmail.com.

1 INTRODUÇÃO

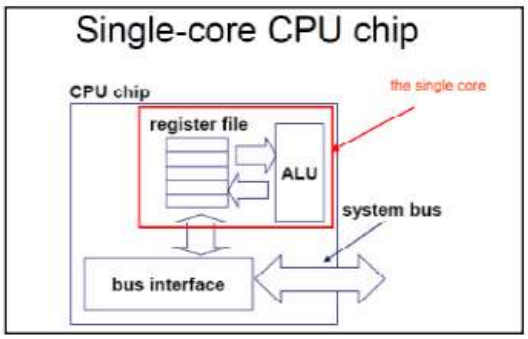
Segundo as observações de Gordon Moore em 1965, quando o mesmo trabalhava como Diretor de Pesquisa e Desenvolvimento na empresa Fairchild Semiconductor, o número de transistores em um microchip dobraria a cada ano. (MOORE, 1998). Essa afirmação, publicada em um artigo chamado em 1965 ficou conhecida como “Lei de Moore” e por muitos anos se provou como fundamental para a operação de múltiplas empresas de tecnologia, entre elas a Intel, empresa que Moore é co-fundador.

A sua observação fala sobre a capacidade de engenharia de fabricação dos chips. Cada vez que era diminuído o tamanho dos transistores era possível aumentá-los em quantidade, porém essa capacidade de diminuição e tamanho é finita e eventualmente chegará a um limite. (JAGTAP, 2015).

Conforme a tecnologia dos processadores foi progredindo, o tamanho de seus transistores diminuiu de forma significativa. Eles ficaram tão pequenos e numerosos que se tornou muito difícil aumentar o clock pela limitação física, principalmente pelo superaquecimento gerado. (OLIVEIRA, 2018).

Até aquele momento, as arquiteturas eram simples (Figura 1) e o aumento da eficiência computacional estava limitado pelo desenvolvimento tecnológico, principalmente pelo fato de os processadores precisarem aguardar o termino de uma tarefa antes de iniciar outra. (TANEBAUM, 2010).

Figura 1: ilustra as arquiteturas simples

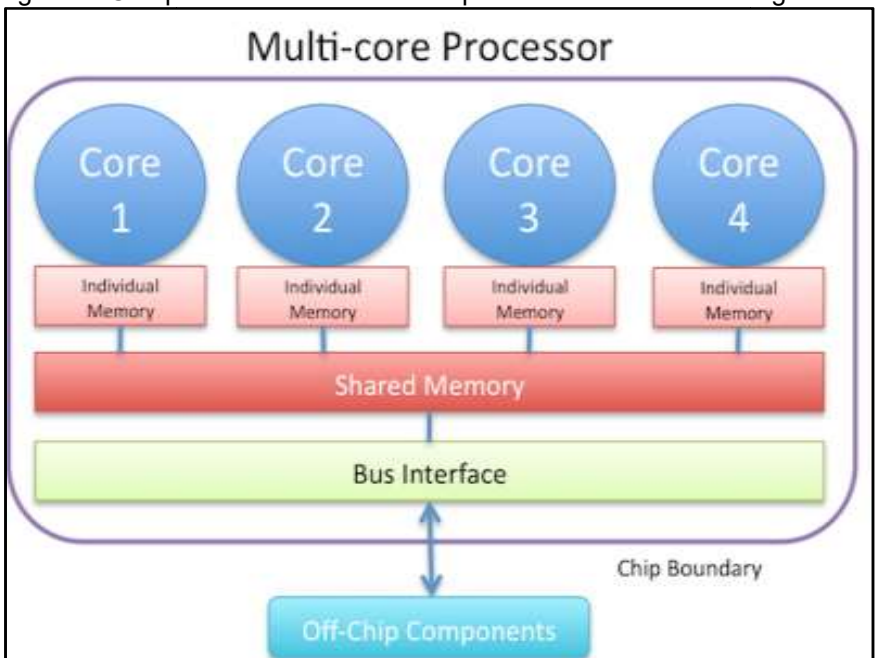


Fonte:International Journal of Advanced Research in Computer Science & Technology

A principal solução para esse problema veio com o uso de mais de um núcleo ao mesmo tempo, através da tecnologia multinúcleos. Processadores multinúcleos são arquiteturas paralelas de memória compartilhada que possuem, em

um único chip de processador, várias unidades de processamento que compartilham a mesma memória.

Figura 2: Componentes básicos de um processador multinúcleos genérico.



Fonte: International Journal of Advanced Research in Computer Science & Technology

Segundo Sanders e Kandrot (2011), o advento dos processadores multinúcleos nos últimos anos foi o resultado da busca pelo poder de processamento. Em processadores desktop melhorias começaram a ser feitas apontando os mesmos para o caminho do *multithreading*.

Essa tecnologia, inicialmente desenvolvida pela Intel com o nome de Hyper-threading, foi lançada no Pentium 4. A mesma permitia que um processador físico se comportasse como dois processadores lógicos, com duas threads de execução. Hoje já temos processadores domésticos com até 16 núcleos completos e 32 threads. (TANENBAUM, BOS, 2015).

Embora a tecnologia multinúcleos esteja presente em praticamente todos os computadores atuais, o desenvolvimento de aplicações paralelas não é uma tarefa simples necessitando de extenso conhecimento em programação e hardware. Nesse cenário, a Open Multi Processing (OpenMP) é uma interface de programação de aplicativo (API) que simplifica a tarefa de paralelizar algoritmos pois ela funciona a base de diretivas de programação que são colocadas diretamente nos blocos de código, não necessitando alteração da lógica do algoritmo ou conhecimento sobre hardware. A mesma está disponível nas linguagens Fortran, C e C++.

Essa pesquisa se propõe a avaliar o desempenho de cinco algoritmos desenvolvidos na linguagem C++ e paralelizados com a OpenMP. Serão avaliados os ganhos utilizando as métricas: tempo de execução, aceleração e eficiência em relação e o esforço necessário para paralelização dos algoritmos.

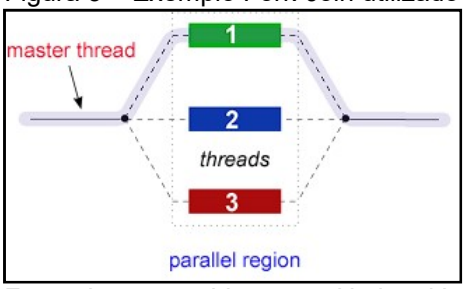
2 OPENMP

A OpenMP é uma API voltada a programação paralela criada com o intuito de padronizar o desenvolvimento em sistemas de memória compartilhada, já que no início eram os fabricantes que forneciam as extensões e justamente por ser designado por cada fabricante individualmente acabavam sendo criadas implementações diferentes que tinham o mesmo objetivo (SILVA, 2001).

A mesma consiste em um grupo de diretivas de compilação, biblioteca de rotinas e variáveis de ambiente que fornecem ao desenvolvedor total domínio do que deve ser paralelizado. Apenas conhecendo algumas diretivas, é possível a um desenvolvedor paralelizar um código, já que as diretivas não mudam a lógica do código, apenas apontam ao compilador qual a região a ser paralelizada.

O modelo de paralelismo da OpenMP é feito exclusivamente via threads, utilizando o modelo *fork-join*, sendo que “uma thread é uma sequencia única de instruções que pode ser executada paralelamente com outras threads.” (JUNIOR, 2017). Nesse modelo, todos os programas iniciam utilizando uma única thread, que é considerada a thread “mestre”. Ao encontrar no código uma região paralela é realizado um fork, ou seja, a criação de novas threads. Quando termina o trecho paralelo apontado pela diretiva, as threads são terminadas, ficando apenas a thread mestre novamente até que a mesma encontre uma nova região paralela ou o programa chegue ao fim de sua execução conforme exemplifica a figura 3.

Figura 3 – Exemplo Fork-Join utilizado na OpenMP



Fonte: Lawrence Livermore National Laboratory (2021)

A implementação da paralelização na API é feita via diretivas de compilação, disponíveis para ser usado em C/C++ ou Fortran, combinando trechos sequenciais e paralelos no mesmo código fonte. O número de threads em uma determinada região pode ser determinado pelo usuário, contudo, em programas paralelos utiliza-se normalmente uma thread para cada núcleo de processamento da arquitetura (CHAPMAN, JOST, PAS, 2008).

2.1 DIRETIVAS DE COMPILAÇÃO

As diretivas de compilação da OpenMP são instruções especiais de pré-processamento que são inseridas no código e interpretadas pelo compilador. As diretivas em C e C++ iniciam com *#pragma*. Para que o compilador as execute, é necessário declarar a biblioteca *omp.h* no cabeçalho do programa e incluir o comando *−fopenmp* na hora de compilar o programa.

Essas diretivas são colocadas diretamente nos trechos que serão paralelizados e o compilador vai ter o trabalho de interpretá-las e paralelizar o código, ou simplesmente ignorá-las caso o compilador não tenha suporte a OpenMP. No caso da linguagem C++, utilizada nessa pesquisa, a sintaxe segue o seguinte padrão: Diretiva *#pragma omp*, seguido por uma palavra chave utilizada para identificação da diretiva, e então as cláusulas separadas por vírgula.

Para exemplificar o funcionamento, utilizaremos o famoso programa “Olá Mundo” conforme exemplifica a Figura 4. Nele, foi incluso a biblioteca *omp.h*, o *#pragma omp* seguido da diretiva *parallel* indicando que aquele trecho do código vai ser paralelizado. A cláusula *num_threads(4),* configura o compilador para que utilize quatro threads nessa execução.

Figura 4 – Exemplo de código em C++

```
#include <iostream>
#include <omp.h>

int main()
{

    #pragma omp parallel num_threads(4)
    {
        std::cout << " Ola Mundo! \n";
    }
}
```

Fonte: Do autor

O código foi compilado duas vezes, a primeira sem o comando *–fopenmp* para que o compilador ignore as diretivas e execute de forma sequencial, a segunda com o comando, habilitando a paralelização do código.

Como resultado da primeira compilação, sem o comando *–fopenmp* temos a frase “Ola Mundo!” uma única vez, já que o compilador ignorou as instruções de paralelismo. O resultado da segunda compilação, será a impressão da frase por cada uma das threads apontadas na clausula *num_threads(4),* ou seja, quatro vezes.

Esse exemplo serve para ilustrar que com pequenas adições no código podemos paralelizar o mesmo sem ter que reescrevê-lo e que mesmo que o compilador não suporte a API, a compilação do algoritmo não será impedida.

2.2 CLAUSULAS

Algumas cláusulas disponibilizadas pelo OpenMP (OPENMP, 2020).

a) *num_threads*: Especifica o número de threads para uma região de paralelização;
b) *schedule*: Define como as tarefas do loop serão escalonadas;
c) *nowait*: Substitui uma barreira implícita na diretiva.

2.3 ESCALONADOR

A cláusula *schedule* define como as tarefas serão divididas e executadas entre as threads dentro de um laço *FOR*, sendo definido o seu tipo (*kind*) e o tamanho dos blocos de tarefas por thread (*chunk*) da seguinte forma: ”*Schedule (kind[,chunk])”.*

Existem cinco tipos de escalonadores: *static, dynamic, guided, runtime* e *auto*, porém apenas os 3 primeiros possuem *chunk*. Caso a cláusula *schedule* não esteja explícita na diretiva, ela assumirá o valor padrão, que é o tipo *static* com o *chunk* igual ao número de iterações dividido pelo número de threads (PACHECO, 2011).

3 MATERIAIS E MÉTODOS

Esta é uma pesquisa aplicada, de base tecnológica. Ela objetivou demonstrar através das métricas tempo de execução, aceleração e eficiência, o ganho em algoritmos paralelizados utilizando a OpenMP, considerando o esforço necessário para a paralelização dos mesmos.

Foi para testes um processador Core i5 4590 com quatro núcleos/threads, sistema operacional Linux Mint 20.3 e compilador g++. Todos os códigos foram compilados usando o comando *–fopenmp*, que habilita o compilador a utilizar/entender as diretivas do OpenMP. Exemplo: g++ -o <programa compilado> -fopenmp <programa.c>.

Para demonstração dos resultados foram utilizados cinco algoritmos que foram paralelizados utilizando a OpenMP. Cada algoritmo foi executado 10 vezes e feito uma média dos resultados. Esse número de iterações foi escolhido baseado na pesquisa bibliográfica, conforme Balaki e Kartheeswaran (2018) que utilizaram 5 repetições e Araujo, Waber, Puntel, Charão e Lima (2018) que utilizaram 5 e 10 repetições.

O programa serial é o mesmo programa paralelo, rodado em uma thread sendo a quantidade de threads foi configurada utilizando a variável de ambiente OMP_NUM_THREADS na execução do programa. Exemplo para o programa de Multiplicação de matrizes: OMP_NUM_THREADS=4 ./Matriz 1000.

A tabela 1 mostra os parâmetros utilizados em cada teste.

Tabela 1 – Parâmetros utilizados

Algoritmo	Parâmetros	Tamanhos		
Decomposição LU	Tamanho da Matriz	1000	2000	3000
Fractal de Mandelbrot	Número de Iterações	100000	300000	500000
Multiplicação de Matrizes	Tamanho da Matriz	1000	1500	2000
Problema da Mochila	Quantidade de livros	150000	150000	150000
	Capacidade da Mochila	1000	2500	5000
Quicksort	Quantidade Números	1000000	5000000	10000000

Fonte: Elaborado pelo Do autor

3.1 MULTIPICAÇÃO DE MATRIZES

A multiplicação de matrizes é um problema matemático da área de álgebra linear em que corresponde ao produto entre duas matrizes. A multiplicação de matrizes é um ótimo exemplo para demonstração de processamento paralelo, devido a demanda de tempo computacional que é utilizado (BALAJI, KARTHEESWARAN, 2018).

A paralelização do programa, conforme mostra figura 5, foi feita tanto na inicialização das matrizes quanto nos laços de multiplicação e soma dos elementos. Foi utilizado o escalonador configurado como static, que é o modo padrão da OpenMP. Nesse modo as iterações são feitas em blocos de tamanhos iguais em todas as threads e a divisão e distribuição dos dados é feita antes da execução da área paralela.

Figura 5 – Multiplicação de matrizes – Trecho paralelizado

```
#pragma omp parallel shared(A,B,C) private(i,j,k)
{
   #pragma omp for schedule (static)
   for (int i = 0; i < TAM; i++) {
      A[i]  = new float[TAM];
   }
   #pragma omp for schedule (static)
   for (int i = 0; i < TAM; i++) {
      B[i]  = new float[TAM];
   }

   #pragma omp for schedule (static)
   for (int i = 0; i < TAM; i++) {
      C[i]  = new float[TAM];
   }

   #pragma omp for schedule (static)
   for (i = 0; i < TAM; i++) {
      for (j = 0; j < TAM; j++) {
         for (k = 0; k < TAM; k++) {
            C[i][j] += A[i][k] * B[k][j];
         }
      }
   }
}
```

Fonte: Do autor

3.2 PROBLEMA DA MOCHILA

O problema da mochila é um problema de análise combinatória que consiste em dado uma quantidade de livros, sendo que os mesmos têm um determinado peso e valor, deve-se preencher uma mochila com o maior valor possível sem extrapolar a capacidade de peso máximo da mesma.

No algoritmo usado nesta pesquisa, é utilizada a abordagem dinâmica com backingtrack. Nessa abordagem é montada uma matriz com número de /capacidade. Cada linha é calculada utilizando a linha anterior, sendo que a primeira

linha consiste nos valores máximos. A segunda linha adiciona o segundo objeto para o cálculo e compara com a primeira.

A paralelização foi feita a partir do segundo FOR, já que o primeiro é necessário para controle da linha anterior. Foi utilizado o escalonador configurado como static, conforme mostra a Figura 6.

Figura 6 – Problema da Mochila – Trecho paralelizado

```
int i = 1;
for (i = 1; i < numLivros; i++)
{
    const int wi = vecPesoLivros[i];
    #pragma omp parallel shared(vecTotal, vecValorLivros, vecPesoLivros)
    {
        #pragma omp for private(cap) schedule(static)
        for (cap = 0; cap <= capMochila; cap++)
        {
            int novoValor;
            if (cap < vecPesoLivros[i])
            {
                totalLivros[cap] = totalLivrosAnterior[cap];
                continue;
            }
        novoValor = vecValorLivros[i] + totalLivrosAnterior[cap - wi];
        totalLivros[cap] = (totalLivrosAnterior[cap] >= novoValor
                            ? totalLivrosAnterior[cap] : novoValor);
        }
    }
 totalLivrosAnterior = totalLivros;
 totalLivros += (capMochila + 1);
 }
```

Fonte: Do autor

3.3 DECOMPOSIÇÃO LU

O algoritmo para resolver sistemas lineares utilizando uma variação do método de Gauss-Seidel, é um processo iterativoe requer a decomposição da matriz principal em duas matrizes triangulares: L (lower) inferior e U (upper) superior.

A aplicação LU utiliza o método Symmetric Successive Over-Relaxation (SSOR), que funciona combinando duas execuções do algoritmo Successive Over-Relaxation (SOR).

A etapa do algoritmo para obter as matrizes triangulares inferior e superior (Decomposição LU), foi implementado para execução paralela, utilizada o escalonador configurado como static.

A implementação consiste em encapsular o método SSOR em uma região paralela. O trecho de código encapsulado é anotado com o pragma omp for para computação paralela. O uso das diretivas distribui a eliminação de cada linha da matriz, resolvendo de forma mais eficiente (Figura 7).

Figura 7 – Decomposição LU – Trecho paralelizado

```
int i = 1;
for (i = 1; i < numLivros; i++)
{
    const int wi = vecPesoLivros[i];
    #pragma omp parallel shared(vecTotal, vecValorLivros, vecPesoLivros)
    {
        #pragma omp for private(cap) schedule(static)
        for (cap = 0; cap <= capMochila; cap++)
        {
            int novoValor;
            if (cap < vecPesoLivros[i])
            {
                totalLivros[cap] = totalLivrosAnterior[cap];
                continue;
            }
            novoValor = vecValorLivros[i] + totalLivrosAnterior[cap - wi];
            totalLivros[cap] = (totalLivrosAnterior[cap] >= novoValor
                                ? totalLivrosAnterior[cap] : novoValor);
        }
    }
totalLivrosAnterior = totalLivros;
totalLivros += (capMochila + 1);
}
```

Fonte: Do autor

3.4 QUICKSORT

O Quicksort é um algoritmo que ordena uma cadeia elementos. O algoritmo testado utiliza uma implementação iterativa do Quicksort, combinando quicksort e insertsort.

É passado ao programa o número de elementos a serem ordenados e um valor de parâmetro que serve para o algoritmo escolher a abordagem de ordenação a ser escolhida. O algoritmo utiliza uma pilha com os intervalos a serem ordenados, quando o intervalo de elementos é menor que o parâmetro, ele utilizada a função de insertsort, caso o contrario utiliza Quicksort.

Os trechos paralelizados são a chamada da função, conforme Figura 8. Nos acessos a pilha, foi configurado com o pragma *critical*, que permite apenas uma thread acessando por vez.

Figura 8 – Quicksort – Trecho paralelizado

```
#pragma omp parallel shared(myVec, globalTodoStack, numThreads, \
    switchThresh, numBusyThreads, globalStackWrite)
{
    if (0 == omp_get_thread_num()) {
        myQuickSort(myVec, 0, myVec.size() - 1, switchThresh,
                    globalTodoStack, numBusyThreads, numThreads,
                    globalStackWrite);
    } else {
        myQuickSort(myVec, 0, 0, switchThresh, globalTodoStack,
                    numBusyThreads, numThreads, globalStackWrite);
    }
}
```

Fonte: Do autor

3.5 FRACTAL DE MANDELBROT

Segundo Santos (2018, p 15), um fractal é uma figura geométrica que cada parte da mesma se assemelha com o todo. No conjunto de Mandelbrot, quando atribuído um valor utilizado a formula $z_{n+1} = z_n^2 + c$ onde z e c são números complexos, a seqüência gerada não tente ao infinito, mas forma uma padrão.

O algoritmo testado calcula a Fractal de Mandelbrot e a renderiza utilizando caracteres ASCII, calculando vários pixels em paralelo. Nele foi utilizado o escalonador configurado como *Dynamic*, onde a carga de dados das threads não é fixa, mas sim decidida em tempo de execução pelo compilador. Também foi utilizada clausula *ordered*, sendo que a mesma força que o cálculo seja feito antes da impressão (Figura 9).

Figura 9 – Fractal de Mandelbrot – Trecho paralelizado

```
#pragma omp parallel for ordered schedule(dynamic)
for(int pix=0; pix<num_pixels; ++pix)
{
    const int x = pix%largura, y = pix/largura;

    complex c = inicio + complex(x * span.real() / (largura +1.0),
                                 y * span.imag() / (altura+1.0));

    int n = MandelbrotCalculate(c, maxiter);
    if(n == maxiter) n = 0;

  #pragma omp ordered
    {
      char c = ' ';
      if(n > 0)
      {
          static const char charset[] = ".,c8M@jawrpogOQEPGJ";
          c = charset[n % (sizeof(charset)-1)];
      }
      std::putchar(c);
      if(x+1 == largura) std::puts("|");
    }
}
```

Fonte: Do autor

3.6 METRICAS

Para a comparação dos algoritmos foram utilizadas as métricas: Tempo de execução, Aceleração e Eficiência.

O tempo de execução é medido em segundos e milissegundos, sendo o tempo calculado dentro do próprio algoritmo.

A aceleração indica quantas vezes um algoritmo paralelo é mais rápido que o sequencial (FONTES, SOUZA, NETO, SILVEIRA, 2014). É calculado com a formula tempo sequencial/tempo paralelo.

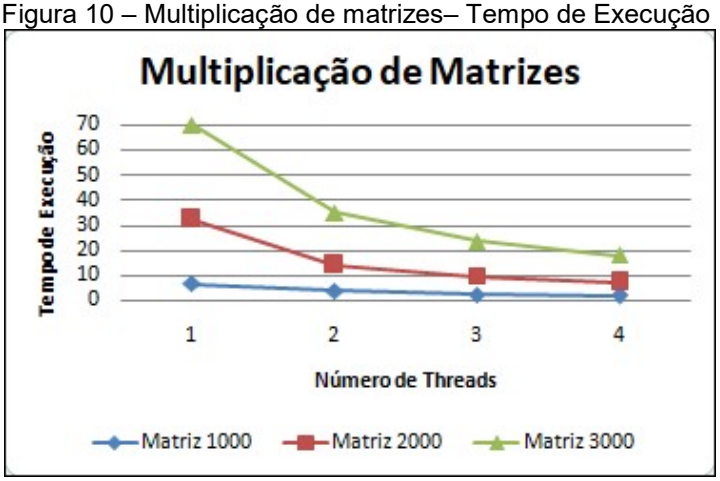
A eficiência visa medir proporção de recursos disponíveis sendo utilizados pelo algoritmo em sua execução. Em processamento paralelo, é esperado que o

desempenho cresça na mesma proporção do número de processadores, mas nem sempre é possível devido a limitações como escalabilidade, comunicação entre outros (AMORIM, OLIVEIRA, FREITAS, 2015). É calculada dividindo a aceleração pelo numero de threads utilizadas.

4 RESULTADOS E DISCUSSÃO

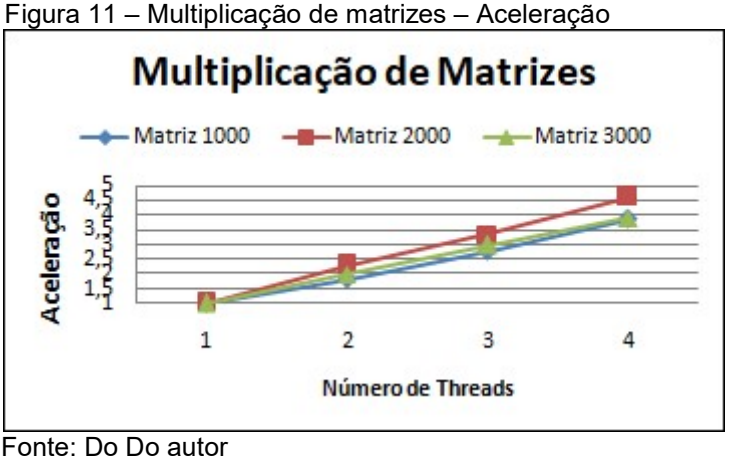
Analisando os dados apresentados nos testes do algoritmo Multiplicação de Matrizes, é possível verificar claramente os ganhos na paralelização do mesmo conforme figuras 10, 11 e 12.

Figura 10 – Multiplicação de matrizes– Tempo de Execução



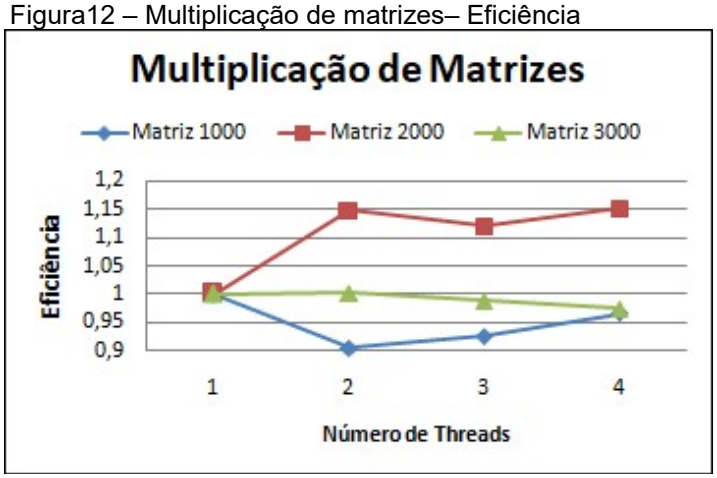
Fonte: Do Do autor

Figura 11 – Multiplicação de matrizes – Aceleração



Fonte: Do Do autor

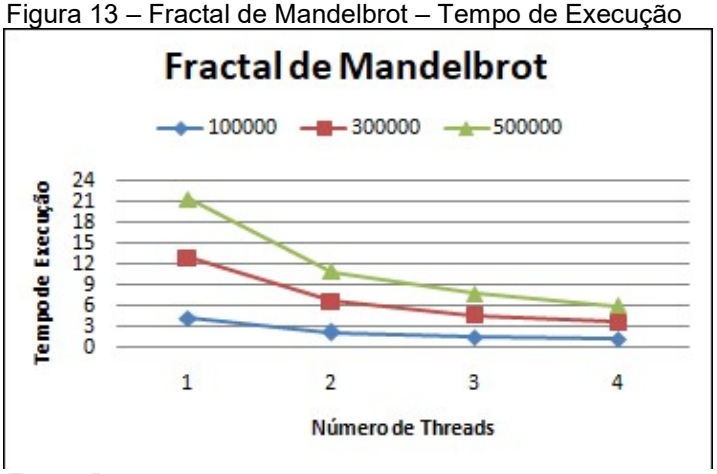
Figura12 – Multiplicação de matrizes– Eficiência



Fonte: Do Do autor

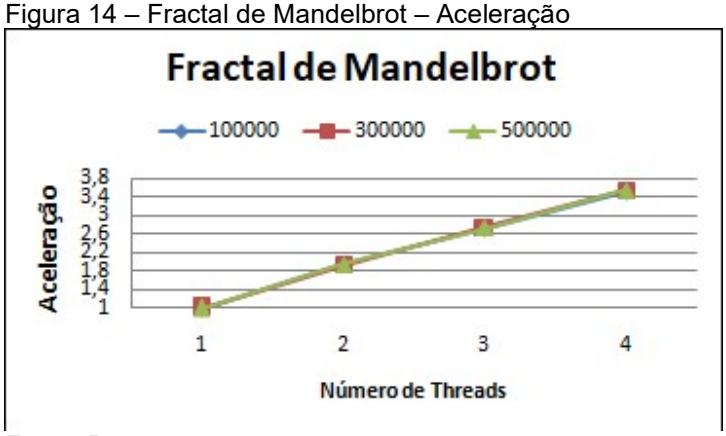
No algoritmo da Fractal de Mandelbrot, a redução no tempo de execução clara (Figura 13) e observa-se que independente do número de iterações utilizadas à aceleração e eficiência são muito próximas quando comparadas por número de threads (Figuras 14 e 15). A eficiência cai consideravelmente conforme mais threads são adicionadas.

Figura 13 – Fractal de Mandelbrot – Tempo de Execução



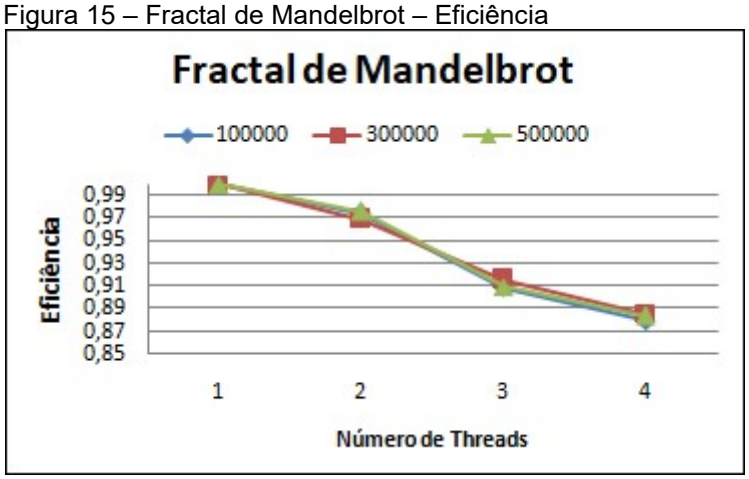
Fonte: Do autor

Figura 14 – Fractal de Mandelbrot – Aceleração



Fonte: Do autor

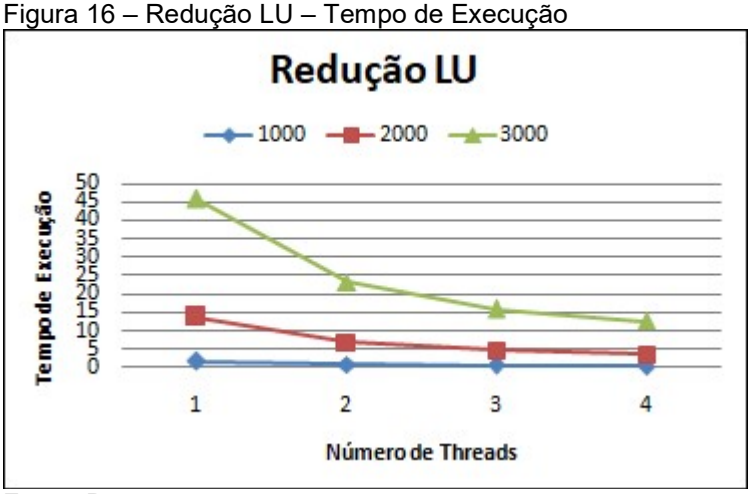
Figura 15 – Fractal de Mandelbrot – Eficiência



Fonte: Do autor

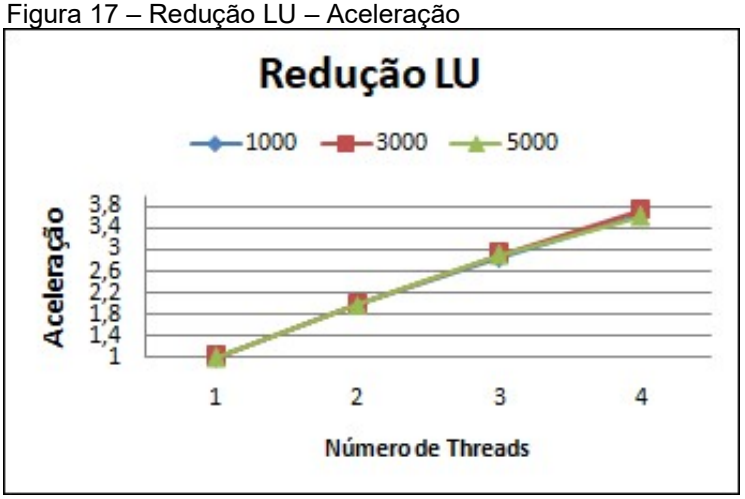
Os testes do algoritmo Redução LU mostra ganho em tempo de execução (Figura 16) e aceleração e eficiência muito próximos independente do tamanho da matriz e quantidade de threads (Figuras 17 e 18).

Figura 16 – Redução LU – Tempo de Execução



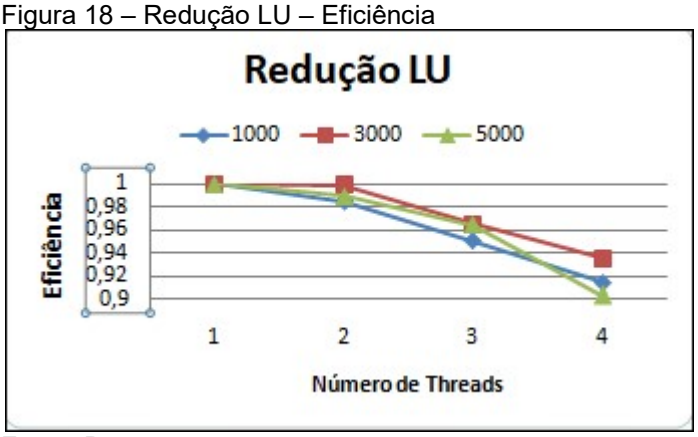
Fonte: Do autor

Figura 17 – Redução LU – Aceleração



Fonte: Do autor

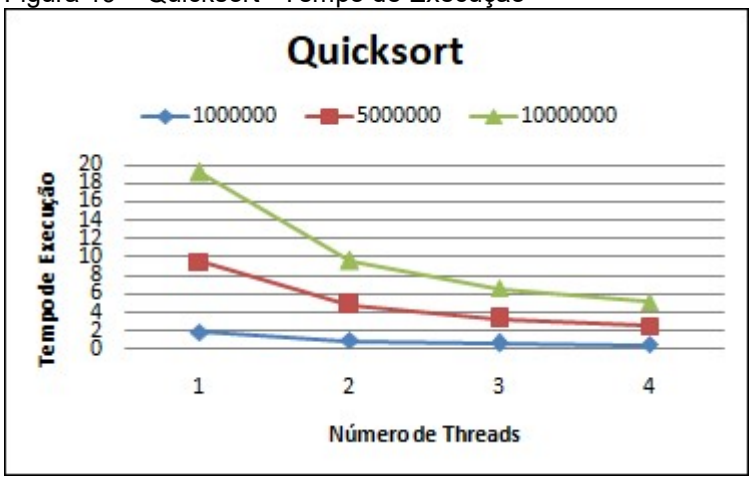
Figura 18 – Redução LU – Eficiência



Fonte: Do autor

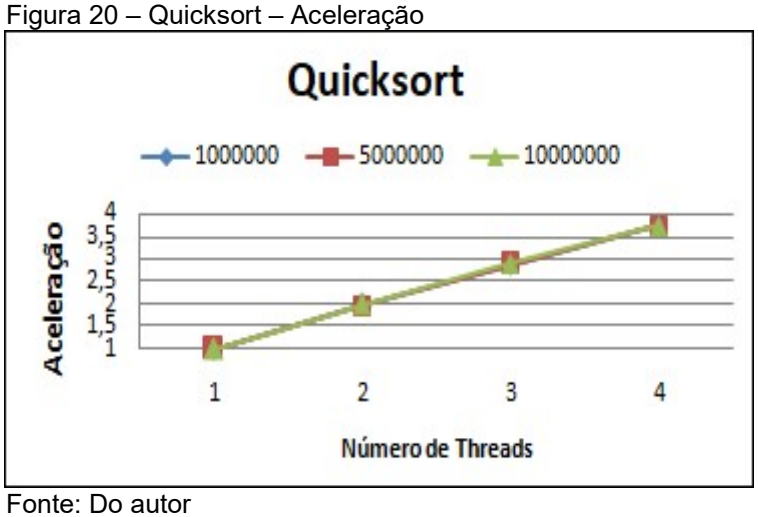
No algoritmo Quicksort, ganhos em tempo de execução conforme figura 19. Tanto aceleração quanto eficiência muito parecidos independente do tamanho de dados utilizado (Figuras 20 e 21).

Figura 19 – Quicksort– Tempo de Execução



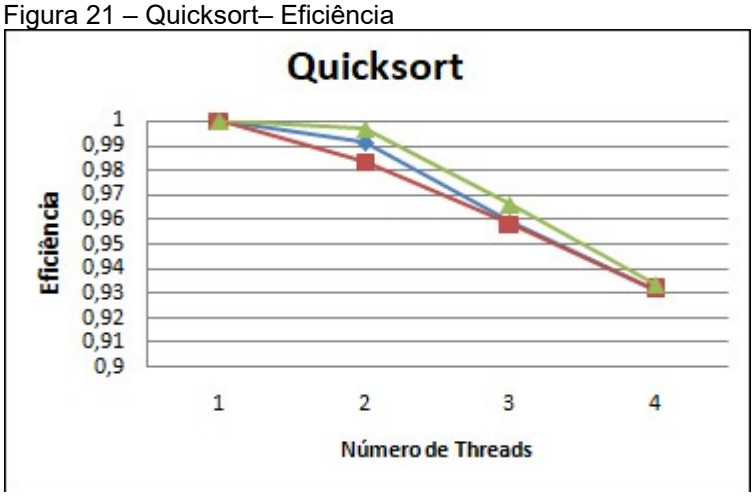
Fonte: Do autor

Figura 20 – Quicksort – Aceleração



Fonte: Do autor

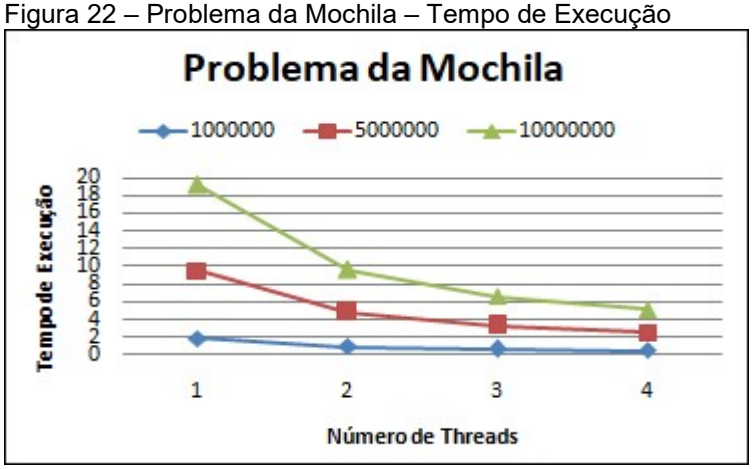
Figura 21 – Quicksort– Eficiência



Fonte: Do autor

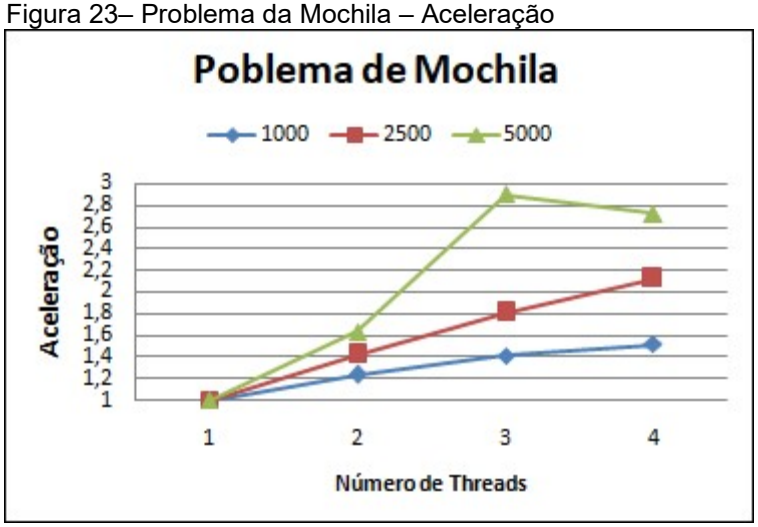
No Problema de Mochila bons ganhos em tempo de execução e aceleração conforme figuras 22 e 23. Em termos de eficiência (Figura 24), temos o pior resultado entre os algoritmos testados.

Figura 22 – Problema da Mochila – Tempo de Execução



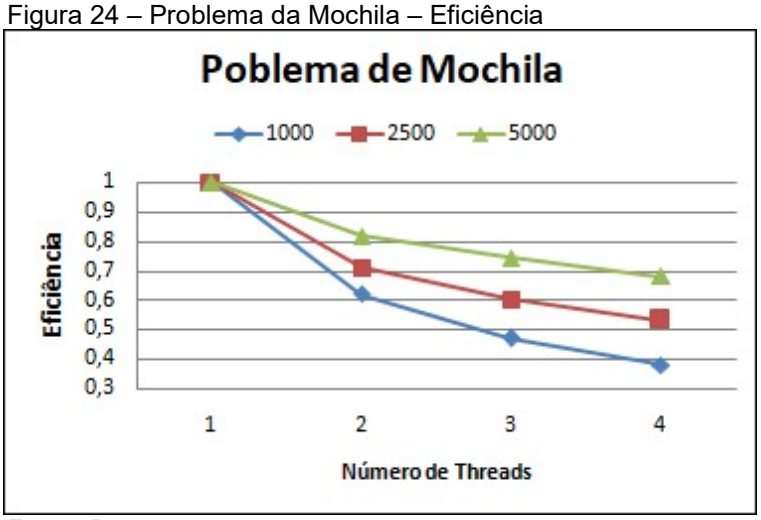
Fonte: Do autor

Figura 23– Problema da Mochila – Aceleração



Fonte: Do autor

Figura 24 – Problema da Mochila – Eficiência



Fonte: Do autor

A Tabela 2 mostra o esforço em linhas de código utilizado para a paralelização de cada algoritmo.

Tabela 2 – Linhas de código referentes a OpenMP

Algoritmos	Linhas de código adicionadas
Decomposição LU	2
Fractal de Mandelbrot	3
Multiplicação de Matrizes	6
Problema da Mochila	3
Quicksort	4

Fonte: Do autor

5 CONCLUSÃO

Nesta pesquisa foram avaliados cinco algoritmos implementados na linguagem C++ paralelizados utilizando a API OpenMP, analisando o ganho de desempenho e esforço empregado na paralelização dos mesmos.

Foi avaliado que a OpenMP alcança o objetivo para o qual é proposta, onde fica evidente o ganho significativo de desempenho em todos os algoritmos testados, principalmente considerando esforço em alteração no código. Mesmos em casos onde a eficiência do algoritmo paralelo foi baixa, como o Quicksort, ainda temos um ganho considerável em tempo de execução, que justifica a o esforço empregado na paralelização do mesmo. Apesar da simplicidade da implementação, a API mostra-se robusta com uma documentação extensa e inúmeras possibilidades

de configurações, o que da ao desenvolvedor uma margem para otimização caso ele queria extrair o máximo de paralelismo do algoritmo.

Conclui-se também que apenas adicionar as diretivas sem um devido estudo do algoritmo pode ter resultados contrários ao esperado, podendo comprometer o ganho ou até piorar o desempenho do mesmo. Usando a OpenMP, o desenvolvedor pode se preocupar menos com a implementação e focar mais no estudo problema em si, considerando que nem todo problema totalmente paralelizável.

Para pesquisas futuras recomenda-se uma analise de desempenho da OpenMP comparando-a com a OpenACC, uma API semelhante a OpenMP que se propõe a facilitar a programação para Unidade de Processamento Gráfico (GPU).

REFERÊNCIAS

AL_DABBAGH, S. S. M.; BARNOUTI, N. Parallel Quicksort Algorithm using OpenMP. **International Journal of Computer Science and Mobile Computing**, v 5, n. 6, p. 372-382., 2016. Disponivel em: Acesso em: 19 out. 2021.

AMORIM, A. M. P.; OLIVEIRA, A. C.; FREITAS, H. C.; Performance evaluation of single- and multi-hop wireless networks-on-chip with NAS Parallel Benchmarks. **Journal of the Brazilian Computer Society.** V. 21, n.6 2015. Disponível em https://journal-bcs.springeropen.com/articles/10.1186/s13173-015-0027-y> Acesso em 07 jun. 2022.

BALAJI, C. K.; KARTHEESWARAN, S. Analyzing the Matrix Multiplication Performance in Shared memory processor Under Multinúcleos Architecture Using OpenMP. **International Journal of Pure and Applied Mathematics**, v. 119, n. 15, p. 3249-3256, 2018. . Disponível em: <https://acadpubl.eu/hub/2018-119-15/2/343.pdf> Acesso em: 20 mar. 2022.

CHAPMAN, B.; JOST, G.; PAS, R. **Using OpenMP: Portable Shared Memory Parallel Programming.** Ed. 1. Cambridge: MIT Press, 2007. p.21-25

CHHIBBER, R.; GARG, R. B. Multinúcleos Processor, Parallelism and Their Performance Analysis. **International Journal of Advanced Research in Computer Science & Technology**. v. 2, n. 3, p. 31-37, 2014. Disponível em: <http://ijarcst.com/doc/vol2-issue3/ver.1/rakhee_chhibber.pdf> Acesso em 4 jun. 2022.

FONTES, A. R.;SOUZA. S. X.;NETO, A. D. D.;SILVEIRA, F. Q. S. On the parallel efficiency and scalability of the correntropy coefficient for image analysis. **Journal of the Brazilian Computer Society**, 2014. Disponível em < https://journal-

bcs.springeropen.com/articles/10.1186/s13173-014-0018-4 > Acesso em 07 jun. 2022.

GRAEFF, M. O. **Solução Paralela para um sistema de Roteirizarão utilizando o problema do Caixeiro Viajante.** 2020. 72f. Trabalho de Conclusão de Curso (Bacharel em Ciência da Computação) – Universidade de Caxias do Sul, Caxias do Sul. Disponível em: <https://repositorio.ucs.br/xmlui/bitstream/handle/11338/9707/TCC%20Cesar%20Augusto%20Graeff.pdf?sequence=1&isAllowed=y > Acesso em: 27 set. 2021.

JAGTAP, M. P. Era of multi-core processors. **Defense Researsh and Development Organization Science Spectrum**. v. 2, n.1 p. 87-94, 2018. Disponível em: <https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.470.9304&rep=rep1&type=pdf> Acesso em: 06 jun. 2022.

LAWRENCE LIVERMORE NATIONAL LABORATORY. **OpenMP Programming Model**. Disponível em: <https://hpc-tutorials.llnl.gov/openmp/programming_model> Acesso em: 3 Mar. 2022.

MAROWAKA, A. PARALLEL COMPUTING ON ANY DESKTOP. **Comunications of the ACM**, v. 50, n.9, p.74-78, 2007. Disponível em: <https://cacm.acm.org/magazines/2007/9/5572-parallel-computing-on-any-desktop/fulltext> Acesso em: 18 ago. 2021

MOORE, G.E; Cramming More Components Onto Integrated Circuits. **Eletronics**, v. 38, N. 8, p.114-117, 1965. Disponível em: <https://newsroom.intel.com/wp-content/uploads/sites/11/2018/05/moores-law-electronics.pdf> Acesso em 06 mai. 2022.

OLIVEIRA, L. P. de. **Uso de Computação Paralela para Acelerar a Cripto-Compressão de Dado.** 2018. 72f. Trabalho de Conclusão de Curso (Bacharel em Ciência da Computação) – Universidade Federal de Santa Catarina, Florianópolis Disponível em: <https://repositorio.ufsc.br/bitstream/handle/123456789/192176/Monografia.pdf> Acesso em: 06 jun. 2022.

OPENMP. **OpenMP Application Programming Interface**. Disponível em: < https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf> Acesso em: 01 mar. 2022.

PACHECO, P. **An Introduction to Parallel Programming.** 1. ed. Burlington: Morgan Kaufmann Publishers, 2011. 209-241 p.

Patterns for Efficient Computation. 1. ed. Burlington: Morgan Kaufmann Publishers, 2012. 21 p.

SANTOS, J.J dos. **Um Breve Estudo Sobre os Fractais: Contruções, Conceito e Aplicações**. 2018. 72f. Trabalho de Conclusão de Curso (Licenciatura em Matemática) – Universidade Estadual do Sudeste da Bahia, Vitória da Conquista. Disponivel em: <http://www2.uesb.br/cursos/matematica/matematicavca/wp-

content/uploads/TCC-JANFFERSON-JOS%C3%89-DOS-SANTOS-VERS%C3%83O-FINAL-Com-corre%C3%A7%C3%B5es-3.pdf> Acesso em: 15 Jun. 2022.

SILVA, M de. O. **Controle de Granularidade em Tarefas OpenMP**, 2011. 72f. Trabalho de Conclusão de Curso (Bacharel em Ciência da Computação) – Universidade do Rio Grande do Sul, Rio Grade do Sul.) Disponível em: <https://www.lume.ufrgs.br/bitstream/handle/10183/36925/000819155.pdf> Acesso em: 27 set. 2021.

TANENBAUM, A. S.; BOS, H. **Modern Operating Systems**. 4. ed. New Jersey: Pearson, 2015. 26 p.

WAZLAWICK, Raul Sidnei**. Metodologia de Pesquisa para Ciência da Computação**. 3.ed. Rio de Janeiro: LTC, 2021.

ARAUJO, L, R de; WEBER, C, M; PUNTEL, F, E; CHARÃO, A, S; LIMA, J, V, F; Análise Comparativa de MPI e OpenMP em Implementações de Transposição de Matrizes para Arquitetura com Memória Compartilhada. **Edição Especial: Artigos do Workshop de Iniciação Científica em Arquitetura de Computadores e Computação de Alto Desempenho (WSCAD/WIC)** . V. 16, N.5 p. 4, 2018 Disponível em: <https://seer.ufrgs.br/index.php/reic/article/view/87684/50666> Acesso em: 07 Jul. 2022.