UNIVERSIDADE FEDERAL DE SANTA CATARINA
CAMPUS TRINDADE

**Leandro Perin de Oliveira**

# USO DE COMPUTAÇÃO PARALELA PARA ACELERAR A CRIPTO-COMPRESSÃO DE DADOS

**FLORIANÓPOLIS**
**2018**

LEANDRO PERIN DE OLIVEIRA

# USO DE COMPUTAÇÃO PARALELA PARA ACELERAR A CRIPTO-COMPRESSÃO DE DADOS

**Trabalho de Conclusão de Curso submetido à Universidade Federal de Santa Catarina, como requisito necessário para obtenção do Grau de Bacharel em Ciências da Computação.**
**Orientador: Prof. Dr. Márcio Bastos Castro**

Florianópolis
2018

# RESUMO

Algoritmos de compressão e cifragem de dados eficientes são bastante necessários no atual cenário da computação. Serviços executados na *web* precisam processar grandes quantias de dados, como imagens e vídeos, e entregar rapidamente o resultado aos usuários.

Neste trabalho são propostas otimizações de desempenho para o algoritmo de cripto-compressão de dados desenvolvido pela *Sheffield Hallam University* no Reino Unido, o qual foi originalmente proposto para compactar dados de um sistema de reconhecimento de rostos em 3D em sistemas de segurança aeroportuária. Foram consideradas duas versões simplificadas do algoritmo original: uma textos e outra para imagens.

As otimizações de desempenho propostas são baseadas na reescrita do código original em C++ com auxílio de funções nativas do C++ e também no uso de técnicas de paralelização de código. A paralelização proposta faz uso da interface de programação OpenMP e, portanto, possui foco em as arquiteturas *multicore* com memória compartilhada.

Os resultados obtidos mostraram ganhos de desempenho de até 11,9x na versão do algoritmo para textos e de até 2,54x na versão do algoritmo para imagens em comparação com as versões originais. Ao final do trabalho pôde-se obter um algoritmo eficaz e eficiente, que comprime os dados em uma taxa muito boa, garante a segurança das informações e executa com maior velocidade.

**Palavras-chave**: Criptografia. Análise de desempenho. Computação de alto desempenho. Sistemas paralelos. Compressão de dados.

# Lista de ilustrações

# Lista de tabelas

# Siglas

**AES** *Advanced Encryption Standard*. 14, 26, 31, 34

**ALU** *Arithmetic Logic Unit*. 20

**API** *Application Programming Interface*. 17, 20, 39, 55

**BMP** *Bitmap*. 33

**CPU** *Central Processing Unit*. 17, 18, 20, 47, 50

**CUDA** *Compute Unified Device Architecture*. 20, 55

**DCT** *Discrete Cosine Transform*. 33, 35, 36, 43, 44

**DES** *Data Encryption Standard*. 26

**DH** *Diffie-Hellman Key Exchange*. 27

**ELGAMAL** *ElGamal Encryption System*. 27

**FPGA** *Field-programmable Gate Array*. 14

**GIF** *Graphics Interchange Format*. 13, 29

**GMPR** *Geometric Modelling and Pattern Recognition Group*. 13, 14, 15, 31, 33, 39, 48, 50, 51, 52, 55

**GPGPU** *General Purpose Graphics Processing Unit*. 20

**GPU** *Graphics Processing Unit*. 20, 47, 55

**JPEG** *Joint Photographic Experts Group*. 13, 29, 33

**JPEG2000** *Joint Photographic Experts Group 2000*. 29, 33

**LZ** *Lempel Ziv*. 29

**LZW** *Lempel Ziv Wech*. 29

**MD5** *Message Digest*. 27

**MIMD** *Multiple Instruction, Multiple Data*. 17, 18

**MISD** *Multiple Instruction, Single Data*. 17, 18

**MIT** *Massachusetts Institute of Technology*. 26, 27

**MPI** *Message Passing Interface*. 17, 20, 55

**NORMA** *Non-Remote Memory Access*. 20

**NSA** *National Security Agency*. 27

**NUMA** *Non-Uniform Memory Access*. 19, 47, 48, 51

**OPENCL** *Open Computing Language*. 20, 55

**OPENCV** *Open Source Computer Vision*. 35, 36, 43, 55

**OpenMP** *Open Multi-Processing*. 14, 20, 21, 23, 24, 39, 40, 41, 42, 47, 55

**PNG** *Portable Network Graphics*. 13, 29

**RLE** *Run Lenght Encoding*. 29

**RSA** *Rivest-Shamir-Adleman*. 26

**SHA** *Secure Hash Algorithm*. 27

**SIMD** *Single Instruction, Multiple Data*. 17, 20

**SIMT** *Single Instruction, Multiple Threads*. 20

**SISD** *Single Instruction, Single Data*. 17

**TIFF** *Tagged Image File Format*. 13, 29

**UMA** *Uniform Memory Access*. 19

# Sumário

# 1 Introdução

Serviços como o YouTube, Instagram, Facebook, dentre outros, possuem uma enorme quantidade de dados armazenados, incluindo texto, imagens e vídeos. Com o aumento no uso de serviços como esses, maior tráfego de dados através da rede e necessidade de armazenamento cada vez maiores, são requeridos métodos mais eficientes para compressão de imagens e vídeos, com alta qualidade de reconstrução e redução na quantidade de armazenamento necessária para os dados [Rodrigues e Siddeq 2016].

Atualmente existem diversos algoritmos que comprimem imagens. Alguns garantem fidelidade máxima, chamados de compressão sem perdas, que é o caso dos formatos *Portable Network Graphics* (PNG) e *Tagged Image File Format* (TIFF), enquanto outros acabam perdendo parte da imagem original, chamados de compressão com perdas, que é o caso dos formatos *Joint Photographic Experts Group* (JPEG) e *Graphics Interchange Format* (GIF). Vale mencionar que esses algoritmos não fazem uso de criptografia, eles apenas comprimem as imagens [Salomon 2007].

Portanto, a disponibilidade de um algoritmo de compressão e cifragem de dados mais eficiente, que consiga reduzir mais o tamanho dos arquivos, comprimir e criptografar de forma rápida, e ainda assim mantendo um bom desempenho na recuperação dos dados é algo que melhoraria bastante o uso de serviços com alto tráfego de informações. Tal algoritmo tornaria os serviços em nuvem mais populares, afinal deixaria os mesmos mais rápidos e seguros, encorajando o desenvolvimento de cada vez mais aplicativos que fazem uso de muitos dados [Stallings 2014].

Um exemplo de um sistema que necessita de algoritmos eficientes para criptografia e compressão de dados pode ser encontrado em aeroportos do Reino Unido. Um sistema de reconhecimento de rostos em 3D foi desenvolvimento para o uso em aeroportos do Reino Unido, o problema é que o mesmo estava gerando uma quantia significativa de dados, aproximadamente 20MB para cada rosto em 3D, sendo duas capturas para cada pessoa, uma no check-in e outra no portão de embarque. Para exemplificar, um voo com 400 passageiros iria gerar 8GB de dados, que precisariam ser enviados para a Polícia Metropolitana de Londres e para a polícia do local de destino, e num aeroporto que tem um fluxo de aproximadamente 200 mil passageiros por dia, gerar 4TB de dados diariamente causaria um grande problema para trocar informações com celeridade.

Pesquisadores da *Sheffield Hallam University* desenvolveram um algoritmo de cripto-compressão chamado *Geometric Modelling and Pattern Recognition Group* (GMPR) para resolver o problema de enviar imagens 3D com grande quantidade de informações de forma segura [Rodrigues e Siddeq 2016]. O algoritmo de compressão desenvolvido

apresentou resultados impressionantes em contraste com os formatos 3D tradicionais. Assim surgiu a ideia de estender o código para comprimir imagens, texto e outros formatos de dados.

Eventualmente foi pensado que o mesmo poderia ser utilizado em aspectos de segurança, como uma forma de criptografia. A conclusão foi de que apenas o cabeçalho dos arquivos seriam criptografados com o conhecido algoritmo *Advanced Encryption Standard* (AES), sendo o corpo dos arquivos apenas comprimidos, sendo impossível visualizar os dados corretamente por conta do cabeçalho estar cifrado com o AES.

A ideia dos pesquisadores da *Sheffield Halam University* é continuar o desenvolvimento do algoritmo, implementando o mesmo em *Field-programmable Gate Arrays* (FPGAs), acelerando o processo e focando em comprimir máquinas virtuais completas.

Embora os algoritmos de cripto-compressão mencionados funcionem corretamente, eles ainda demoram um tempo considerável para processar conteúdos grandes, como imagens de alta resolução ou textos com milhares de linhas, o que torna inviável o uso dos mesmos em aplicações que exigem resposta rápida aos usuários.

Este TCC irá trabalhar em uma versão mais simplificada do GMPR, focada em textos e imagens 2D.

## 1.1 Objetivos

Com base no exposto, são apresentados a seguir o objetivo geral e os objetivos específicos do presente projeto.

### 1.1.1 Objetivo Geral

O objetivo geral deste TCC é propor e implementar versões paralelas eficientes dos algoritmos de cripto-compressão GMPR desenvolvidos pela *Sheffield Hallam University* para *multicores*. A solução proposta permitirá reduzir significativamente o tempo de execução dos algoritmos de cripto-compressão.

### 1.1.2 Objetivos Específicos

Os objetivos específicos são listados a seguir:

- Produzir um código sequencial limpo e organizado, das versões de texto e de imagens, do algoritmo GMPR em C++ com base na implementação existente (não otimizada) em C [Rodrigues e Siddeq 2016];

- Propor uma solução paralela para os algoritmos para arquiteturas *multicore* com uso de *Open Multi-Processing* (OpenMP);
- Realizar experimentos com o intuito de medir o desempenho das soluções propostas em uma plataforma *multicore*.

## 1.2 Justificativa

Este trabalho se insere em uma colaboração inicial entre o Laboratório de Pesquisa em Sistemas Distribuídos (LaPeSD) da UFSC e a *Sheffield Hallam University* (Reino Unido), proponente e desenvolvedora do algoritmo GMPR. A necessidade de paralelizar o código veio de seus problemas de desempenho, o que inviabiliza o uso em aplicações reais. Esse algoritmo funcionando de forma eficiente e entregando rapidamente o resultado aos usuários é algo que iria reduzir problemas de armazenamento e segurança dos dados, pois poderia cifrar e comprimir os mesmos, recuperando-os corretamente no futuro, sem se tornar um gargalo na aplicação. Este trabalho poderá, também, ser integrado facilmente em projetos futuros que tenham restrições de infraestrutura, necessidade de maior velocidade de processamento ou maior segurança das informações, dentre outras razões.

Este TCC permitirá estudar as melhores técnicas de computação paralela que possam auxiliar no desenvolvimento do trabalho proposto. Tais técnicas serão utilizadas no projeto, implementadas e testadas para que se possa escolher uma ou mais que se mostrem adequadas para o desenvolvimento do trabalho.

## 1.3 Organização do Texto

O texto deste trabalho será organizado da seguinte forma. A Seção 2 apresenta e descreve a fundamentação teórica deste trabalho. A Seção 3 apresenta o algoritmo de cripto-compressão GMPR, incluindo suas principais ideias e pseudo-código. Então, a Seção 4 apresenta algumas ideias e possibilidades de paralelização do algoritmo GMPR para arquiteturas *multicore*. A Seção 5 mostra os resultados obtidos e uma análise dos mesmos. Por fim, as principais observações a respeito do trabalho desenvolvido, possíveis melhorias e usos deste trabalho em projetos futuros são descritas na Seção 6.

# 2 Fundamentação Teórica

## 2.1 Computação Paralela

Há muito tempo o mercado de tecnologia vem buscando cada vez mais velocidade de processamento. Várias áreas demandam muito poder computacional para executar suas tarefas, desde o sequenciamento de DNA até simulações do universo, e para isso necessitam cada vez mais de um maior desempenho.

Antigamente, o aumento de desempenho estava diretamente atrelado ao aumento da frequência dos processadores. Porém, este aumento atingiu um limite físico para a velocidade do relógio. Quando mais rápida a frequência de relógio, menor deve ser o processador, pois os sinais elétricos devem ir e voltar dentro do mesmo ciclo de relógio. A solução para isso poderia ser a construção de uma *Central Processing Unit* (CPU) cada vez menor, porém isso causa problemas de aquecimento, afinal uma maior frequência de relógio gera mais calor. Com a redução do tamanho da CPU torna-se mais complicada a dissipação de calor da mesma.

Executar várias tarefas em paralelo, com várias CPUs trabalhando em conjunto, acabou sendo a forma encontrada para aumentar o poder computacional das máquinas. Hoje em dia já é possível encontrar arquiteturas paralelas com milhares de CPUs. Essas arquiteturas podem ser classificadas em dois grandes grupos: multiprocessadores e multicomputadores. No primeiro grupo (multiprocessadores) encontram-se as arquiteturas compostas por diversas CPUs interligadas através de um barramento ou similar, permitindo assim um compartilhamento da memória principal entre todas as CPUs. Por outro lado, no segundo grupo (multicomputadores), a memória principal é distribuída. A programação paralela nessas arquiteturas é feita através do uso de linguagens de programação, como o Erlang, e *Application Programming Interfaces* (APIs), como o *Message Passing Interface* (MPI), desenvolvidos especialmente para computação paralela [Tanenbaum e Bos 2015].

### 2.1.1 Arquiteturas Paralelas

Arquiteturas paralelas podem ser classificadas segundo seu fluxo de instruções e fluxo de dados utilizando a Taxonomia de Flynn [Flynn 1972]. A Tabela 1 mostra as quatro classes possíveis de arquiteturas segundo esta classificação.

**SISD:** Um único fluxo de instruções trabalha em um único fluxo de dados. É a representação clássica de uma arquitetura sequencial. A Figura 1 ilustra essa classe.

**SIMD:** Uma única instrução é responsável pelo processamento de vários dados.

| | Único | Múltiplo |
|---|---|---|
| Único | *Single Instruction, Single Data* (SISD) | *Single Instruction, Multiple Data* (SIMD) |
| Múltiplo | *Multiple Instruction, Single Data* (MISD) | *Multiple Instruction, Multiple Data* (MIMD) |

Tabela 1 – Taxonomia de Flynn


Figura 1 – Classe SISD da Taxonomia de Flynn

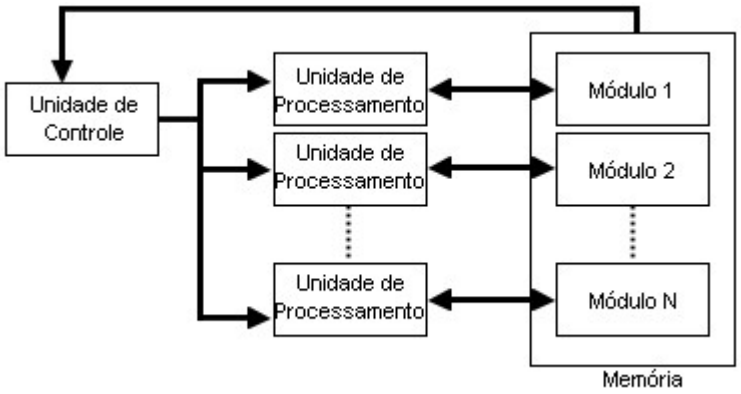



Figura 2 – Classe SIMD da Taxonomia de Flynn

Define o funcionamento de processadores vetoriais e matriciais. Diversos módulos de memória são necessários, as instruções seguem organizadas sequencialmente e possui uma unidade de controle e várias unidades de processamento. A Figura 2 ilustra essa classe.

**MISD:** Múltiplas instruções trabalhando no mesmo fluxo de dados. Esta classe da Taxonomia de Flynn é impossível de ser colocada em prática. A Figura 3 ilustra essa classe.

**MIMD:** Múltiplas instruções trabalhando em múltiplos dados. Possui várias unidades de controle, várias unidades de processamento e vários módulos de memória. Qualquer grupo de máquinas operando em conjunto, com interação entre elas, pode ser classificado como MIMD. A Figura 4 ilustra essa classe.

Além da Taxonomia de Flynn, as arquiteturas paralelas podem ser classificadas segundo o compartilhamento de memória. Multiprocessadores trabalham com vários processadores podendo acessar a mesma memória compartilhada, enquanto nos multicomputadores cada processador possui sua própria memória, fazendo necessário o uso de uma rede de interconexão para trocar informações.

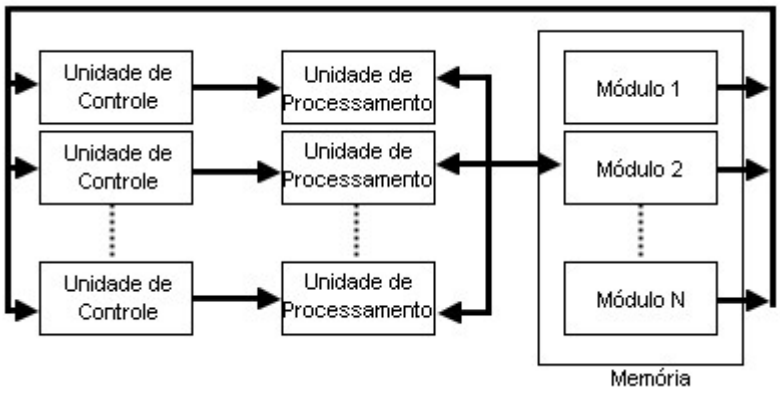



Figura 3 – Classe MISD da Taxonomia de Flynn

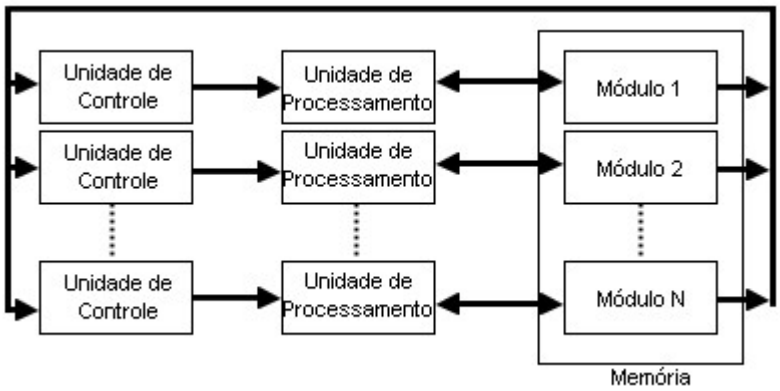



Figura 4 – Classe MIMD da Taxonomia de Flynn

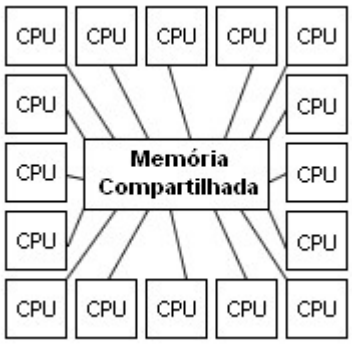



Figura 5 – Multiprocessador

#### 2.1.1.1 Multiprocessadores

Multiprocessadores são sistemas nos quais múltiplas CPUs compartilham acesso à mesma memória. Uma propriedade que forma a base da comunicação entre processadores é: uma CPU escreve algum dado na memória e outra lê o mesmo dado. Sistemas multiprocessadores possuem algumas características únicas, como sincronização de processos e escalonamento, por exemplo. A Figura 5 exibe graficamente a arquitetura de um multiprocessador.

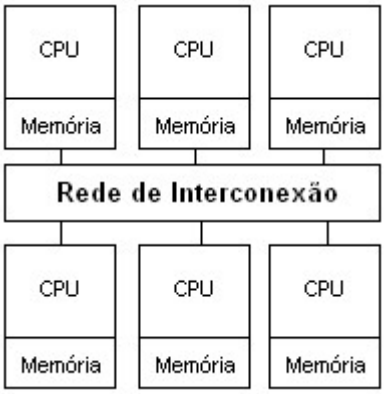



Figura 6 – Multicomputador

Alguns multiprocessadores possuem a característica de que uma certa palavra de memória possa ser lida na mesma velocidade que qualquer outra palavra. Essas máquinas são chamadas de *Uniform Memory Access* (UMA). Máquinas que não apresentam essa propriedade são chamadas de *Non-Uniform Memory Access* (NUMA) [Tanenbaum e Bos 2015].

#### 2.1.1.2 Multicomputadores

Multicomputadores são sistemas nos quais cada CPU possui sua própria memória, não podendo ser diretamente acessada por nenhum outro processador. A troca de informações nesse sistema é feita através de uma rede de interconexão. A Figura 6 exibe graficamente a arquitetura de um multicomputador.

O acesso à memória nos multicomputadores é classificado como *Non-Remote Memory Access* (NORMA), afinal não é possível que uma CPU tenha acesso à memória remota [Hwang e Xu 1998].

#### 2.1.1.3 Aceleradores

Uma *General Purpose Graphics Processing Unit* (GPGPU), ou um acelerador, é uma placa gráfica que pode ser usada para computação de propósito geral. São classificados como SIMD pela Taxonomia de Flynn, permitindo que vários dados possam ser processados em paralelo. As placas atuais utilizam uma extensão desse conceito, chamada de *Single Instruction, Multiple Threads* (SIMT), garantindo a execução da mesma instrução em threads diferentes. A Figura 7 exibe graficamente a arquitetura de um acelerador.

GPGPUs são compostas por vários processadores, que possuem vários núcleos cada, sendo que cada processador possui sua própria memória interna, compartilhada pelas suas *Arithmetic Logic Units* (ALUs), e uma memória compartilhada entre todos os processadores [Miranda 2010].

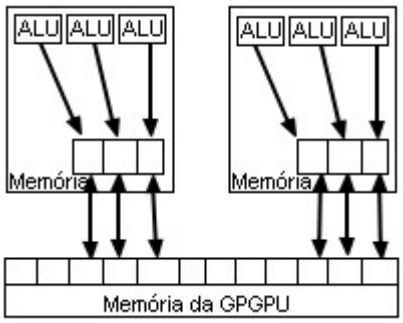



Figura 7 – Acelerador

### 2.1.2 Programação Paralela

As técnicas, linguagens ou APIs para o desenvolvimento de aplicações paralelas estão, em grande parte dos casos, diretamente atreladas às arquiteturas. Por exemplo, para multicomputadores o modelo de programação mais difundido para paralelismo é o MPI. No caso de *Graphics Processing Units* (GPUs) e aceleradores o *Compute Unified Device Architecture* (CUDA) e o *Open Computing Language* (OPENCL) são os mais utilizados. Neste trabalho, iremos focar em arquiteturas de memória compartilhada (multiprocessadores). Para este tipo de arquitetura, uma das APIs mais aceitas e utilizadas é o OpenMP.

O OpenMP utiliza o modelo de paralelismo conhecido como *Fork-Join*. Este modelo, formulado em 1963, propõe que um programa serial poderá criar múltiplas *threads* para executar uma parte do código em paralelo *(Fork)*. A parte do código executada por múltiplas *threads* é denominada neste modelo de **região paralela**. Ao final da execução em uma região paralela, todas as *threads* realizam uma sincronização explícita e, então, a execução do programa segue de maneira serial *(Join)* [McCool, Reinders e Robison 2012].

O OpenMP é baseado em diretivas de compilação, podendo ser usado em C/C++ ou em Fortran, combinando regiões sequenciais e paralelas no mesmo código fonte. As diretivas permitem a criação de regiões paralelas nas quais múltiplas *threads* são criadas e executadas em paralelo. O número de *threads* em uma região paralela pode ser determinado pelo usuário, todavia, em programas paralelos utiliza-se normalmente uma *thread* para cada núcleo de processamento da arquitetura [Chapman, Jost e Pas 2008].

Para C/C++ as diretivas são da seguinte maneira:

```
#pragma omp [diretiva] [atributos]
```

Para iniciar uma região paralela, é necessário utilizar a diretiva `parallel` logo antes do bloco que será paralelizado, da seguinte maneira:

```
#pragma omp parallel
```

O número de *threads* dentro de região paralela pode ser configurado pelo desenvolvedor com auxílio da função `omp_set_num_threads(int num_threads)`, invocando-a antes da região paralela. Uma outra maneira possível é através do uso do atributo `num_threads(int num_threads)` que poderá ser utilizado em conjunto com a diretiva `parallel`. Caso o desenvolvedor não configure o número de *threads*, o OpenMP irá criar uma *thread* para cada núcleo de processamento.

Por padrão, todas as variáveis globais são compartilhadas entre as *threads* de uma região paralela. Porém, variáveis criadas dentro de uma região paralela são consideradas privadas (ou seja, cada *thread* irá possuir uma cópia local dessas variáveis). Para uma variável declarada fora do laço ter um contexto privado para cada *thread*, é necessário o uso do atributo `private` ou `firstprivate`. O primeiro cria uma cópia privada da varável para cada *thread* sem inicializá-la. O seguindo, por outro lado, além de criar uma cópia privada inicializa o valor da cópia com o valor que a variável possuía antes da região paralela.

A Figura 8 mostra um exemplo do uso da diretiva `parallel` e dos atributos `private` e `shared`.

```
1   int vglobal = 42, vlocal = 0;
2
3   #pragma omp parallel private(vlocal) shared(vglobal)
4   {
5       int id = omp_get_thread_num();
6       vlocal = id * vglobal;
7       printf("Thread %d: vlocal = %d\n", id, vlocal);
8   }
9   //vlocal permanece com valor 0 a partir deste ponto
```

Figura 8 – Exemplo de uso dos atributos `private` e `shared`.

Conforme o exemplo mostra, na linha 3 foi usada a diretiva `private` para declarar a variável `vlocal` privada para cada thread, enquanto a variável `vglobal` é compartilhada.

A variável `id` é privada para cada thread, pois foi declarada dentro da região paralela, não precisando da diretiva `private` a declarando explicitamente. Na linha 5 a variável `id` recebe o número da thread que está executando, através da função `omp_get_thread_num()`.

Enquanto a variável `vglobal` possui o mesmo valor para todas as threads, a variável `vlocal` tem um valor diferente para cada thread a partir da execução da linha 6, na qual um valor é atribuído para a mesma. Antes da atribuição do valor a variável não possui nenhum conteúdo, estando vazia.

Após o fim da região paralela na linha 8, a variável `vlocal` permanece com o valor 0, que ela já possuía antes de entrar na região.

A diretiva `for`, quando utilizada em conjunto com a diretiva `parallel`, permite dividir a computação das iterações de um laço entre as *threads* de uma região paralela. Isso pode ser feito de duas maneiras: 1) utilizando-se a diretiva `#pragma omp parallel`

seguida da diretva `#pragma omp for`; ou 2) combinando ambas as diretivas em uma única diretiva da seguinte maneira:

```
#pragma omp parallel for
```

A Figura 9 mostra um exemplo de soma de matrizes utilizando um laço paralelizado com a diretiva `parallel for`. Neste exemplo, as duas *threads* criadas na região paralela dividem a computação das iterações do laço mais externo (linha 3), onde cada *thread* será responsável por computar metade do conjunto de iterações em `i`. Em outras palavras, a primeira *thread* será responsável por executar as iterações `i`=$[0, 500)$; a segunda *thread*, por outro lado, será responsável por executar as iterações `i`=$[500, 1000)$.

```
1   int i, j;
2   #pragma omp parallel for private(i,j) num_threads(2)
3   for (i = 0; i < 1000; i++) {
4       for (j = 0; j < 1000; j++)
5           a[i][j] = b[i][j] + c[i][j];
6   }
```

Figura 9 – Exemplo de uso da diretiva `parallel for`.

O OpenMP ainda oferece um atributo que permite unificar laços aninhados em um único laço, o qual será dividido entre as *threads* se a diretiva `for` for utilizada. Este atributo, denominado `collapse`, recebe como parâmetro a quantidade de laços para serem unificados. O `collapse` é interessante quando deseja-se expandir o espaço de iterações que será paralelizado. Isso permite um melhor paralelismo, tendo em vista que haverão mais iterações para serem distribuídas entre as *threads*.

O OpenMP ainda permite que o desenvolvedor especifique como as iterações de um laço paralelo devem ser atribuídas as *threads* em uma região paralela. Isto é feito através do atributo `schedule(type [,chunk])`, onde `type` define o algoritmo de escalonamento a ser utilizado e `chunk`, o qual é opcional, define o tamanho do bloco de iterações a ser atribuído a cada *thread* no momento de um escalonamento:

```
#pragma omp parallel for schedule(type [, chunk])
```

Os tipos básicos de escalonamento definidos no OpenMP são o `static` (padrão), *dynamic* e *guided*. O escalonador `static` divide igualmente as tarefas entre as *threads* de maneira estática, ou seja, essa distribuição é feita estaticamente antes da execução do laço paralelo. O escalonador `dynamic`, por outro lado, escalona as iterações em tempo de execução sob demanda (ou seja, cada *thread* recebe um novo `chunk` de iterações somente quando ela fica ociosa). Por padrão, o escalonador *dynamic* escalona iterações individualmente (`chunk=1`), mas esse parâmetro pode ser ajustado pelo usuário. Por fim,

o escalonador `guided` é semelhante ao `dynamic`, porém ele escalona as iterações dos laços em blocos que possuem um tamanho grande inicialmente e, ao longo da execução, os tamanhos dos próximos blocos são reduzidos gradativamente (exponencialmente) para evitar um desbalanceamento de carga entre as *threads* [Chandra et al. 2000].

A Figura 10 mostra um exemplo de uso do escalonador `dynamic` com `chunk` de tamanho 1. Nesse caso, as iterações do laço da linha 3 são distribuídas individualmente às *threads* da região paralela em tempo de execução.

```
1   int i, j;
2   #pragma omp parallel for schedule(dynamic, 1) private(i, j)
3   for(i = 0; i < size; i++)
4      for(j = i; j < size; j++)
5        do_work(i, j);
```

Figura 10 – Exemplo de uso do escalonador `dynamic`.

É possível notar neste exemplo que a quantidade de chamadas ao método `do_work` depende diretamente do valor o iterador `i` do laço mais externo. Portanto, o uso do escalonador padrão do OpenMP (`static`) em uma região paralela com `t` *threads* resultaria em `size/t` iterações do laço da linha 3 atribuídas a cada *thread*. Como a quantidade de iterações a serem executadas no laço mais interno depende do valor de `i`, as *threads* acabariam recebendo cargas de trabalho de tamanho significativamente distintos, causando um desbalanceamento de carga. Como a distribuição das iterações nesse caso seria feita de maneira estática, teríamos uma situação onde diversas *threads* estariam ociosas no final da região paralela enquanto aguardam o término das *threads* que ficaram mais sobrecarregadas. O desbalanceamento poderia se tornar ainda maior se considerarmos o fato de que o tempo de computação do `do_work` é imprevisível. Logo, a estratégia `dynamic` se adéqua melhor a este caso, pois o escalonamento das iterações é feito em tempo de execução e sob demanda, balanceando melhor a carga entre as *threads*.

É importante salientar que os escalonadores `dynamic` e `guided` possuem um sobrecusto maior que o escalonador estático, justamente pelo fato do controle do escalonamento ser feito em tempo de execução. Todavia, quando o desbalanceamento de carga é muito grande em uma região paralela, o ganho de desempenho obtido com um melhor balanceamento de carga se sobrepõe ao sobrecusto.

Quando existem vários laços independentes dentro de uma mesma região paralela, é possível utilizar o atributo `nowait` para evitar que uma *thread* ociosa precise esperar um laço acabar para poder executar o próximo laço. A Figura 11 apresenta um exemplo de uso do atributo `nowait`.

Neste exemplo, se uma *thread* terminar de executar o primeiro laço mas ainda houverem outras *threads* no mesmo, ela não irá esperar e começará a execução do segundo laço imediatamente.

```
1   int i, j;
2   #pragma omp parallel private(i, j)
3   {
4       for (i = 0; i < size; i++) {
5           #pragma omp for nowait
6           for (j = 0; j < size; j++)
7               do_work(i, j);
8
9           #pragma omp for nowait
10          for (j = 0; j < size; j++)
11              do_work2(i, j);
12      }
13  }
```

Figura 11 – Exemplo de uso do atributo `nowait`.

O sucesso do OpenMP se deve ao fato de que ele é bem simples de ser usado e, por conta de uma alta aceitação, consegue ser executado em várias plataformas diferentes [Chapman, Jost e Pas 2008].

## 2.2 Criptografia de Dados

Do grego *Kriptos* (oculto) e *Grapho* (escrita), é o nome dado à ciência de codificar e decodificar mensagens. Tem como meta garantir:

- **Autenticação**: Identificar o remetente da mensagem;
- **Integridade**: Não adulteração da mensagem original;
- **Não Recusa**: Remetente não pode negar que enviou a mensagem.

A criptografia pode ser classificada como simétrica ou assimétrica, dependendo de como as chaves de codificação e decodificação são utilizadas. Há também o resumo criptográfico, ou Hash, que é um número pequeno que representa todo um documento [Stallings 2014].

### 2.2.1 Criptografia Simétrica

O conceito mais antigo de criptografia é chamado de criptografia simétrica. Neste modelo a chave que dá acesso à mensagem é a mesma, tanto para codificar como para decodificar a mensagem, e deve permanecer em segredo, por isso é chamada de chave privada. A chave é utilizada para evitar que terceiros tenham acesso à mensagem, mesmo conhecendo o algoritmo utilizado e tendo em mãos a mensagem cifrada. A Figura 12 exibe o funcionamento da criptografia simétrica.

A maior vantagem da criptografia simétrica é sua facilidade de uso e velocidade para executar os algoritmos criptográficos. O problema deste modelo é que a chave usada

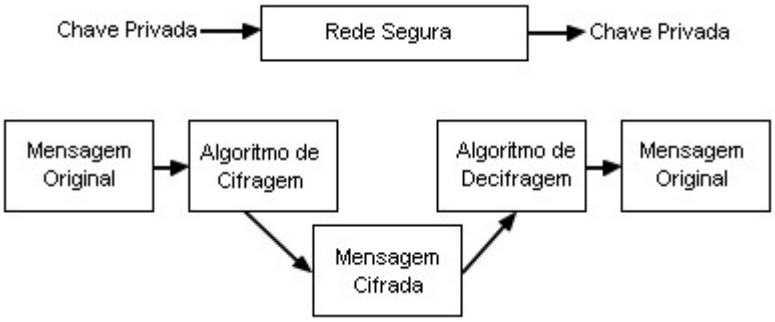



Figura 12 – Criptografia Simétrica

para cifrar precisa ser compartilhada com o destinatário, abrindo uma brecha para terceiros interceptarem a chave [Stallings 2014].

Os principais algoritmos de criptografia simétrica são:

- AES: Desenvolvido pelo *National Institute of Standards and Technology*, é o algoritmo padrão usado pelo governo dos Estados Unidos da América. Possui um tamanho de bloco fixo em 128 bits, chave de 128, 192 ou 256 bits, rápido e fácil de executar e utiliza pouca memória.

- *Data Encryption Standard* (DES): Desenvolvido pela IBM em 1977, foi o algoritmo mais utilizado no mundo até a padronização do AES. Possui um tamanho de chave pequeno, de apenas 56 bits, o que possibilita quebrar o algoritmo por força bruta. A partir de 1993 passou a ser recomendada a utilização do 3DES, uma variação do DES no qual o ciframento é feito 3 vezes seguidas, porém é muito lento para se tornar um algoritmo padrão.

### 2.2.2 Criptografia Assimétrica

Modelo desenvolvido pelo matemático Clifford Cocks, no qual as chaves para cifrar e decifrar são diferentes, chamadas de assimétricas. A chave pública pode ser vista por qualquer pessoa, porém a chave privada permanece em posse apenas do titular. Uma pessoa pode utilizar sua chave privada para decodificar uma mensagem criptografada com sua chave pública. A Figura 13 exibe o funcionamento da criptografia assimétrica.

A maior vantagem deste modelo é a segurança, uma vez que a chave privada não é compartilhada. Porém a velocidade é muito menor do que os algoritmos simétricos, o que pode não permitir o seu uso em algumas situações [Stallings 2014].

Os principais algoritmos de criptografia assimétrica são:

- *Rivest-Shamir-Adleman* (RSA): Desenvolvido em 1977 no *Massachusetts Institute of Technology* (MIT). É o algoritmo assimétrico mais utilizado no momento, além


Figura 13 – Criptografia Assimétrica

de ser um dos mais poderosos que existem. É baseado no fato de que dois números primos são facilmente multiplicados para gerar um terceiro número, porém é muito difícil recuperar esses números a partir do terceiro número. Para se descobrir a chave privada, é necessário fatorar números muito grandes, o que pode levar um tempo considerável. Assim, a segurança do RSA é baseada na dificuldade de fatoração de números primos grandes.

- *ElGamal Encryption System* (ELGAMAL): Baseado em grandes cálculos matemáticos. Sua segurança é baseada na dificuldade de calcular logaritmos discretos em um corpo finito.

- *Diffie-Hellman Key Exchange* (DH): Mais antigo dos métodos assimétricos, também é baseado no problema dos logaritmos discretos. Não é possível usá-lo para assinaturas digitais.

### 2.2.3 Resumo Criptográfico

Resumo criptográfico, também conhecido por hash, são funções criptográficas unidirecionais, ou seja, não é possível obter o conteúdo original a partir do hash. Uma característica dessas funções é que, independente do tamanho do texto, hash sempre terá um tamanho fixo, geralmente de 128 bits. Outra propriedade é que duas mensagens distintas não irão gerar o mesmo hash [Pfleeger, Pfleeger e Margulies 2015].

Os principais algoritmos de hash utilizados atualmente são:

- *Message Digest* (MD5): Desenvolvido por Ron Rivest, do MIT. Produz um hash de 128 bits. É um algoritmo rápido, simples e seguro, porém não é recomendado devido ao pequeno tamanho de 128 bits, sendo preferível um hash de maior valor.

- *Secure Hash Algorithm* (SHA): Criado pela *National Security Agency* (NSA), gera um hash de 160 bits. É recomendável o uso do SHA-2, uma variação mais forte e segura do que o SHA-1, devido ao maior número de bits que é gerado.

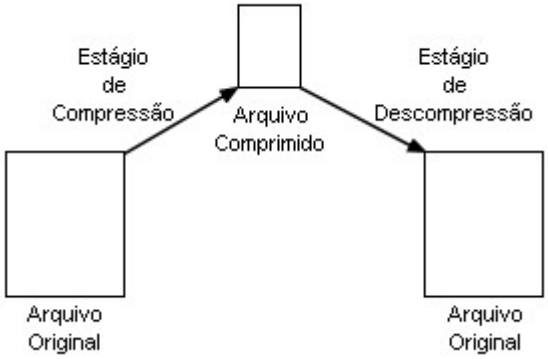



Figura 14 – Compressão de Dados

Alguns usos de funções de hash são:

- **Verificar integridade de arquivos**: Basta tirar o hash de um arquivo e guardá-lo. Em um momento futuro é possível tirar o hash novamente e comparar com o antigo, se forem iguais então o arquivo está íntegro.
- **Armazenamento de senhas**: A forma mais segura de armazenar uma senha é armazenar o hash da mesma, afinal não é possível obter a senha original. Quando precisar ser feita a verificação se uma senha digitada está correta, basta comparar o hash da senha digitada com o hash armazenado, se forem iguais então a senha está correta.

## 2.3 Compressão de Dados

Comprimir dados é o ato de reduzir o tamanho de arquivos, diminuindo o espaço que eles ocupam em disco e aumentando o desempenho de aplicativos que usam esses dados. Essa técnica é interessante para diversos fins, desde um usuário de smartphone que deseja armazenar mais fotos no seu aparelho até um serviço *web* que envia muitos dados através da internet. A Figura 14 mostra o conceito da compressão de dados.

Existem duas formas de compressão, com perdas e sem perdas, que descartam partes insignificantes do arquivo ou mantém todo o conteúdo, respectivamente.

### 2.3.1 Compressão sem Perdas

A compressão sem perdas, ou *Lossless Data Compression*, garante que os dados obtidos após a descompressão serão exatamente iguais aos dados originais que foram comprimidos. Essa técnica é geralmente utilizada quando não se pode perder nada do conteúdo original, como arquivos de texto ou informações delicadas de experimentos científicos por exemplo.

### 2.3.2 Compressão com Perdas

A compressão com perdas, ou *Lossy Data Compression*, garante que os dados obtidos após a descompressão serão bastante parecidos com o conteúdo original, com diferenças mínimas. Essa técnica é geralmente utilizada para arquivos de áudio ou vídeo, nos quais a diferença de conteúdo é imperceptível e o tamanho é reduzido consideravelmente.

### 2.3.3 Compressão de Textos

A compressão de textos se baseia em representar o texto original de outra maneira, usando símbolos que ocupem menos espaço. Com isso se ganha também velocidade ao se fazer busca em grandes documentos. A desvantagem é o tempo necessário para a descompressão do conteúdo.

Um dos métodos mais conhecidos é o método de Huffman, de 1952. Nele, um código único é associado a cada caractere diferente do texto. Códigos menores são associados com os caracteres que aparecem com maior frequência. É o método mais eficiente para compressão de textos em linguagem natural, com 60% de redução no tamanho do arquivo. O método de Huffman elimina todos os espaços entre palavras. No momento da descompressão, a não ser que exista um separador, como uma vírgula, um espaço é inserido entre as palavras [Salomon 2007].

### 2.3.4 Compressão de Imagens

Existem formatos de imagens que comprimem com ou sem perdas. Os mais conhecidos formatos de compressão sem perdas são PNG, *Joint Photographic Experts Group 2000* (JPEG2000) e TIFF.

A compressão sem perdas explora a redundância entre pixeis e garante que nenhum dado será perdido. É especialmente importante em casos nos quais a fidelidade dos dados é muito importante, como para a fotografia profissional. Os algoritmos mais usados são o *Run Lenght Encoding* (RLE), *Lempel Ziv* (LZ), *Lempel Ziv Wech* (LZW) e o algoritmo de Huffman, o qual é usado nos formatos PNG e TIFF.

Dentre os métodos com perdas, os mais conhecidos são JPEG e GIF. A compressão com perdas busca eliminar detalhes que não são perceptíveis ao olho humano. Porém há formatos, como o GIF, que utilizam um grau maior de perda, causando uma degradação grande na imagem.

# 3 O Algoritmo de Cripto-Compressão GMPR

O algoritmo GMPR possui duas implementações distintas, uma com foco em cripto-compressão de texto e outra em imagens, sendo que ambas as versões serão explicadas em maiores detalhes neste capítulo.

## 3.1 Criptografia

Ambas as versões utilizam o mesmo método para cifrar e decifrar o conteúdo. São geradas 3 chaves na etapa de compressão que são necessárias para recuperar o conteúdo original do arquivo. Essas chaves são armazenadas no cabeçalho dos arquivos.

Posteriormente, o cabeçalho é cifrado com o conhecido algoritmo AES, garantindo a segurança do mesmo. O corpo do arquivo não precisa ser cifrado, pois é necessário o conteúdo do cabeçalho para recuperar o conteúdo do corpo, logo o mesmo fica apenas comprimido, sem a necessidade de outra camada de segurança.

O arquivo sem o cabeçalho fica indecifrável, sendo um binário que não pode ser interpretado. A Figura 15 mostra o que é exibido se um arquivo, no caso um texto 'abac', codificado for aberto com algum editor de texto.

A geração das chaves será explicada ao longo deste capítulo, tanto para a versão de textos quanto para a versão de imagens.

Na etapa de descompressão, o cabeçalho do arquivo é decifrado com o AES e posteriormente o conteúdo original é recuperado com o algoritmo GMPR, o qual será detalhado ao longo do capítulo.

## 3.2 Compressão de Texto

### 3.2.1 Compressão

De modo geral, o algoritmo GMPR é iniciado obtendo o texto plano e convertendo-o para `ASCII`. Em seguida é obtida uma lista dos caracteres contidos no arquivo, sem repetir nenhum. Por exemplo: o texto "abac" irá gerar uma lista contendo os caracteres "abc", não incluindo o caractere "a" duas vezes.

Então são geradas 3 chaves aleatórias. A partir disso é gerada uma lista chamada `nCoded`, na qual a cada 3 caracteres do texto é gerado 1 elemento codificado usando as chaves geradas O primeiro caractere é multiplicado pela primeira chave, o segundo

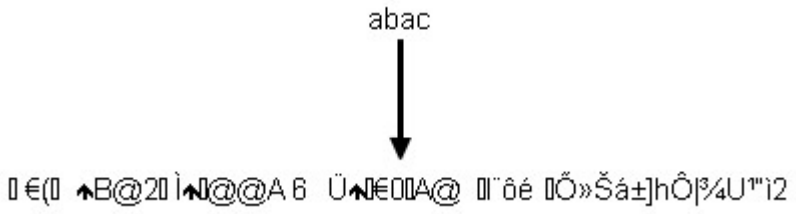



Figura 15 – Binário gerado pelo algoritmo GPMR.

caractere pela segunda chave e o terceiro caractere pela terceira chave, então os 3 valores são somados e adicionados na lista `nCoded`, prosseguindo com os próximos 3 caracteres.

É criado um vetor chamado `codedvector` que é formado pelas 3 chaves, pela lista única de caracteres e pela lista `nCoded`. Em seguida o texto final é gerado. A taxa de compressão é de 1/3 do tamanho original, afinal 1 elemento do texto comprimido representa 3 elementos do texto original.

O Algoritmo 1 apresenta em mais detalhes o processo de compressão de textos. Nas linhas 2 a 4 é criada e preenchida a `nLimited`, uma lista contendo todos os caracteres únicos do texto plano.

Em seguida as 3 chaves aleatórias são geradas nas linhas 5 a 7. `K[0]` é gerada entre 0 e 1000. `K[1]` gera um valor entre 0 e 10 e depois multiplica pelo dobro do maior caractere do texto plano. `K[2]` gera um valor entre 0 e 10, multiplica pela chave `K[1]` e depois multiplica pelo maior caractere do texto plano.

Nas linhas 8 a 10 é criada e preenchida a `nCoded`, uma lista que contém as chaves `K` multiplicadas pelo texto plano. Nas linhas 11 a 15 é criado e preenchido o `codedvector`, um vetor que contém as chaves `K`, o tamanho da lista `nLimited`, o tamanho da lista `nCoded`, o conteúdo da lista `nLimited` e o conteúdo da lista `nCoded`. Na linha 16 é convertido o vetor `codedvector` para `string`.

Na linha 17 é obtida uma lista de intervalos de probabilidades para cada símbolo do texto plano. Em seguida são aplicados os intervalos à cada elemento do texto e adicionados ao texto final codificado nas linhas 18 e 19. Por fim o arquivo é salvo em disco na linha 20.

### 3.2.2 Descompressão

Para descomprimir o conteúdo e obter o arquivo original, o algoritmo começa obtendo o texto cifrado e transformando-o em um vetor. Em seguida são extraídas as 3 chaves usadas no processo de cifragem, a lista única de caracteres e a lista `nCoded`. Por fim, o conteúdo é recuperado.

O Algoritmo 2 apresenta em mais detalhes o processo de descompressão de texto. Na linha 1 a rotina `ArDecodeFile` abre o arquivo cifrado, faz uma leitura do cabeçalho e

**Algorithm 1** Compressão de texto.

```
1:  nData ← abrirArquivoOriginal()
2:  for i from 0 to nData.tamanho() do
3:      if !(nLimited ⊃ nData[i]) then
4:          adicionarElemento(nLimited, nData[i])
5:  K[0] ← rand() % 1001
6:  K[1] ← (rand() % 11) * 2 * maxvalue
7:  K[2] ← K[1] * maxvalue * (rand() % 11)
8:  while i <= nData.tamanho() do
9:      nCoded[i] ← K[0] * nData[i] + K[1] * nData[i + 1] + K[2] * nData[i + 2]
10:     i ← i + 3
11: adicionarElemento(codedvector, K)
12: adicionarElemento(codedvector, nLimited.tamanho())
13: adicionarElemento(codedvector, nCoded.tamanho())
14: adicionarElemento(codedvector, nLimited)
15: adicionarElemento(codedvector, nCoded)
16: string msg ← paraString(codedvector)
17: stats ← obterListaIntervaloProbabilidades()
18: for i from 0 to nData.tamanho() do
19:     AplicaProbabilidades(nData[i], stats)
20: EscreverDisco(stats)
```

gera a lista de probabilidades que será usada para decodificar o arquivo. Em seguida a `string` é convertida para vetor na linha 2.

Nas linhas 3 a 5 são obtidas as 3 chaves que foram usadas para cifrar. O tamanho da lista de caracteres únicos `nLimited` é obtido na linha 6 e seu conteúdo obtido nas linhas 8 e 9. O tamanho da lista `nCoded`, que contém as chaves `K` multiplicadas pelo texto plano, é obtido na linha 7 e seu conteúdo obtido nas linhas 10 e 11.

Nas linhas 12 a 20 o texto é decodificado, obtendo o conteúdo original. É feito o processo inverso da geração da lista `nCoded` no processo de cifragem, e quando o elemento codificado é encontrado então ele é salvo na lista `nDecoded`. Por fim o arquivo é salvo em disco.

## 3.3 Compressão de Imagens

O algoritmo GMPR apresentou uma qualidade superior em relação ao conhecido formato JPEG e qualidade equivalente ao formato JPEG2000. A Tabela 2 compara o tamanho de imagens em *Bitmap* (BMP), JPEG e comprimidas com o GMPR [Rodrigues e Siddeq 2016].

Ele utiliza a *Discrete Cosine Transform* (DCT) e um algoritmo de minimização de matrizes no estágio de compressão, além de um novo método concorrente de busca binária no estágio de descompressão.

**Algorithm 2** Descompressão de texto.

```
msg ← ArDecodeFile()
data ← paraVetor(msg)
K[0] ← data[0]
K[1] ← data[1]
K[2] ← data[2]
nL ← data[3]
nC ← data[4]
for i from 5 to 5 + nL do
    nLimited[i] ← data[i]
for i from 5 + nL + 1 to 5 + nL + 1 + nC do
    nCoded[i] ← data[i]
ultimaPos ← data.tamanho()
for r from 0 to ultimaPos do
    for i from 0 to nL do
        for j from 0 to nL do
            for k from 0 to nL do
                if data[r] == K[0] * nLimited[i] + K[1] * nLimited[j] + K[2] * nLimited[k] then
                    nDecoded[3 * r + 0] ← nLimited[i]
                    nDecoded[3 * r + 1] ← nLimited[j]
                    nDecoded[3 * r + 2] ← nLimited[k]
EscreverDisco(nDecoded)
```

| Imagem | BMP | JPEG | GMPR |
|---|---|---|---|
| Maçã | 336 MB | 52.4 MB | 0.929 MB |
| Estátua | 366 MB | 58.5 MB | 0.916 MB |
| Rosto | 200.7 MB | 45.5 MB | 0.784 MB |

Tabela 2 – Compressão de Imagens.

### 3.3.1 Compressão

Primeiramente a imagem é dividida em blocos de tamanho `n`, e então é aplicada a DCT em cada bloco. Cada bloco consiste de coeficientes `DC`, que são o valor médio do bloco, e os demais coeficientes, chamados `AC`. A DCT, um processo sem perdas e reversível, serve para identificar redundâncias na imagem, ou seja, pixeis muito semelhantes em relação aos seus vizinhos.

Em seguida é aplicado o algoritmo de minimização de matrizes na lista de coeficientes `AC`, eliminando todos os zeros, que geralmente são muitos, reduzindo seu tamanho e produzindo o vetor minimizado. São definidas 3 chaves que irão multiplicar cada 3 entradas deste vetor minimizado, e então os 3 valores são somados, ou seja, cada 3 valores do vetor é transformado em apenas 1, reduzindo o tamanho para 1/3 do original, produzindo o vetor minimizado codificado.

O resultado do processo é um arquivo com muitas informações da imagem original,

porém reduzidas e combinadas, sendo necessário o cabeçalho com as 3 chaves para recuperar a imagem original. Assim, a segurança é garantida com a cifragem do cabeçalho com o algoritmo AES [Rodrigues e Siddeq 2016].

O Algoritmo 3 apresenta em mais detalhes o processo de compressão de imagens. Na linha 1 a imagem é divida em vários blocos de tamanho `n` e posteriormente armazenada em um vetor chamado `img`.

As linhas 2 e 3 são responsáveis por aplicar a DCT em cada bloco do vetor, através do método `cv::dct`, que faz parte da biblioteca *Open Source Computer Vision* (OPENCV).

A linha 4 utiliza o OPENCV com a função `cv::findNonZero`, responsável por eliminar todos os zeros encontrados, e salva o resultado no vetor `nonZeroData`. As linhas 5 a 7 criam e preenchem um vetor contendo a lista única de elementos do vetor sem zeros, sem repetir nenhum.

A linha 8 calcula os intervalos onde estavam localizados os zeros, calculando a diferença entre o vetor sem zeros e o vetor com zeros, armazenando os resultados em outro vetor, o `zeroPosition`.

Em seguida as 3 chaves são geradas nas linhas 9 a 11. `K[0]` é gerada com o valor 1. `K[1]` calcula o maior valor do vetor sem zeros e soma 2. `K[2]` calcula o maior valor do vetor sem zeros e multiplica por 1 somado com a chave `K[1]`.

Nas linhas 12 a 14 é criado e preenchido o vetor `vcoded`, que contém as 3 chaves multiplicadas pelo vetor sem zeros e depois somadas. Nas linhas 15 a 21 é criado e preenchido o vetor `vnSerialized`, que contém as chaves `K`, o tamanho do vetor `vcoded`, o tamanho do vetor `zeroPosition`, o tamanho do vetor `nLimited`, o conteúdo do vetor `vcoded`, o conteúdo do vetor `zeroPosition` e o conteúdo do vetor `nLimited`.

A linha 22 é responsável pela conversão do vetor `vnSerialized` para string. Finalmente o arquivo é salvo em disco na linha 23.

### 3.3.2 Descompressão

A etapa de descompressão inicia na leitura do arquivo codificado e convertendo o mesmo para um vetor. Em seguida são recuperadas as 3 chaves, o vetor sem zeros que representa a imagem, a posição dos zeros removidos e a lista única de caracteres.

É feita a busca binária no vetor sem zeros, combinando as 3 chaves e recuperando assim o conteúdo completo. Em seguida os zeros são colocados novamente nas posições indicadas pelo vetor de posições.

Por último, é realizada a operação inversa da DCT e a imagem original é recuperada e salva em disco [Rodrigues e Siddeq 2016].

**Algorithm 3** Compressão de Imagens.

```
1: imagem[]img ← DIVIDIREMBLOCOS(n)
2: for i from 0 to img.tamanho() do
3:     matDctData[i] ← CV::DCT(img[i])
4: nonZeroData ← CV::FINDNONZERO(matDctData)
5: for i from 0 to nonZeroData.tamanho() do
6:     if !(nLimited ⊃ nonZeroData[i]) then
7:         ADICIONARELEMENTO(nLimited, nonZeroData[i])
8: zeroPosition ← DIFF(matDctData, nonZeroData)
9: K[0] ← 1
10: K[1] ← 2 + MAX_ELEMENT(nonZeroData)
11: K[2] ← MAX_ELEMENT(nonZeroData) * (1 + K[1])
12: while i <= nonZeroData.tamanho() do
13:     vcoded[i] ← K[0] * nonZeroData[i] + K[1] * nonZeroData[i + 1] + K[2] * nonZeroData[i + 2]
14:     i ← i + 3
15: ADICIONARELEMENTO(vnSerialized, K)
16: ADICIONARELEMENTO(vnSerialized, vcoded.tamanho())
17: ADICIONARELEMENTO(vnSerialized, zeroPosition.tamanho())
18: ADICIONARELEMENTO(vnSerialized, nLimited.tamanho())
19: ADICIONARELEMENTO(vnSerialized, vcoded)
20: ADICIONARELEMENTO(vnSerialized, zeroPosition)
21: ADICIONARELEMENTO(vnSerialized, nLimited)
22: string szSerialized ← PARASTRING(vnSerialized)
23: ESCREVERDISCO(szSerialized)
```

O Algoritmo 4 apresenta em mais detalhes o processo de descompressão de imagens. Na linha 1 o arquivo codificado é lido do disco e convertido para um vetor na linha 2.

As linhas 3 a 5 recuperam as 3 chaves utilizadas na codificação. As linhas 6 a 8 recuperam o tamanho dos vetores sem zeros, de posição dos zeros e da lista única de caracteres, respectivamente. As linhas 9 a 14 recuperam o conteúdo desses vetores.

As linhas 15 a 22 decodificam o conteúdo, obtendo o original, através da comparação das chaves com o conteúdo codificado. Na linha 23 é feita a inserção dos zeros que haviam sido removidos, com base na posição indicada pelo vetor `zeroPosition`.

Finalmente, na linha 24 é chamada a função do OPENCV `cv::idct`, que realiza a função inversa da DCT, recuperando a imagem original. O conteúdo é salvo em disco na linha 25.

**Algorithm 4** Descompressão de Imagens.

```
1: szSerialized ← READ_CODED_FILE()
2: data ← PARAVETOR(szSerialized)
3: K[0] ← data[0]
4: K[1] ← data[1]
5: K[2] ← data[2]
6: nZ ← data[3]
7: zP ← data[4]
8: nL ← data[5]
9: for i from 6 to 6 + nZ do
10:     vcoded[i] ← data[i]
11: for i from 6 + nZ + 1 to 6 + nZ + 1 + zP do
12:     zeroPosition[i] ← data[i]
13: for i from 6 + nZ + zP + 1 to 6 + nZ + 1 + zP + nL do
14:     nLimited[i] ← data[i]
15: for r from 0 to vcoded.tamanho() do
16:     for i from 0 to nL do
17:         for j from 0 to nL do
18:             for k from 0 to nL do
19:                 if vcoded[r] == K[0] * nLimited[i] + K[1] * nLimited[j] + K[2] * nLimited[k] then
20:                     nDecoded[3 * r + 0] ← nLimited[i]
21:                     nDecoded[3 * r + 1] ← nLimited[j]
22:                     nDecoded[3 * r + 2] ← nLimited[k]
23: matDctData ← PUSHZEROS(nDecoded, zeroPosition)
24: img ← CV::IDCT(matDctData)
25: ESCREVERDISCO(img)
```

# 4 Paralelização da Cripto-Compressão

O algoritmo descrito no capítulo anterior possui várias possibilidades de ganho de desempenho. Todavia, para que isso fosse possível, uma versão em C++ precisou ser desenvolvida.

Este trabalho de conclusão de curso possui três etapas principais. Na primeira etapa, foi feita uma implementação em C++ do algoritmo de cripto-compressão GMPR, incluindo-se algumas otimizações quando possível. Com base nesta solução proposta, na segunda etapa foram estudadas formas de paralelizar o código para obter-se um desempenho ainda melhor. A API do OpenMP foi explorada em detalhes para implementar soluções paralelas eficientes para este problema. Por fim, na terceira etapa, foram realizados experimentos para medir o desempenho das soluções paralelas propostas. Neste capítulo serão apresentadas as soluções encontradas tanto para a versão de textos quanto para imagens. O código em C++ está disponível no GitHub[1].

Foi usada a API do OpenMP para paralelizar os trechos de código mais custosos (em termos de desempenho) da aplicação em múltiplos núcleos do processador. Conforme visto no Capítulo 3, o algoritmo possui muitos laços do tipo `for`, o que permite a utilização de diretivas da API para executar os mesmos em paralelo, garantindo a correta execução das operações, porém em menos tempo.

Para atingir essa meta, o objetivo inicial foi definir em quais partes do código o programa leva mais tempo para executar, analisar se é possível quebrar essa parte em pedaços pequenos e atribuir múltiplas *threads* para a execução. Para isso, alguns experimentos preliminares foram realizados utilizando-se a ferramenta `Valgrind`[2] no Linux. Os resultados serão mostrados separadamente para as versões de texto e imagens ao longo do capítulo.

Após obter os resultados, o próximo passo foi transformar os setores mais custosos do código em regiões paralelas sempre que possível, a fim de melhorar o desempenho da aplicação. Após definir as regiões, foi necessário analisar a existência de variáveis compartilhadas entre as *threads* e verificar dependências entre elas. As cláusulas `shared` e `private` permitem descrever quais variáveis são compartilhadas entre todas as *threads* e quais são privadas em cada *thread*, respectivamente. Também foi verificada a existência de condições de corrida no código paralelizado. Nesses casos, regiões críticas do código, que são regiões que só podem ser executadas em uma *thread* de cada vez, deverão ser protegidas com o uso da diretiva `omp critical`, que limita a execução pelas *threads*.

[1] <https://github.com/leandroperin/ParallelCryptoCompression>
[2] <http://valgrind.org/>

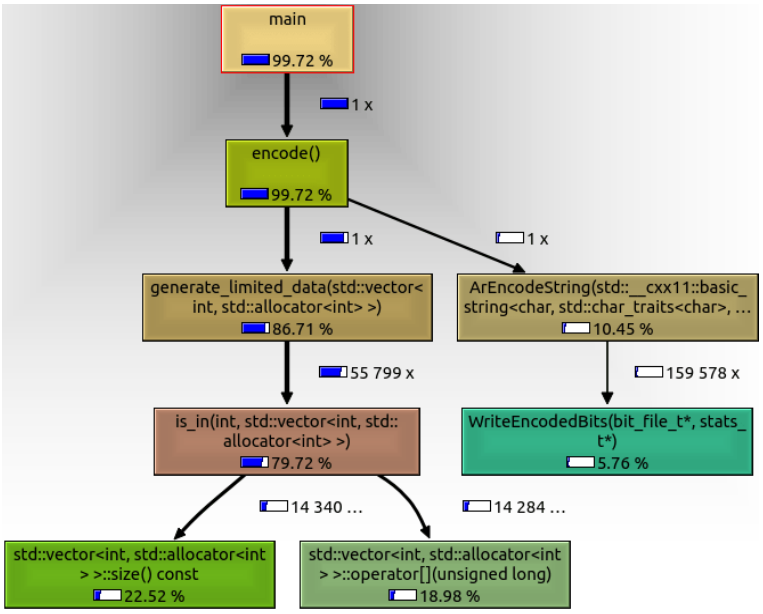



Figura 16 – Percentual do tempo de execução despendido nas principais funções do algoritmo original de codificação de textos.

## 4.1 Cripto-compressão de Texto em Paralelo

O primeiro teste executado foi na execução da codificação de um arquivo de texto. O resultado gerado pelo `Valgrind` é ilustrado na Figura 16. Como é possível observar na Figura 16, o método mais custoso em termos de tempo de execução é o `generate_limited_data()`, o qual utiliza o método `is_in()` para verificar se algum caractere está presente na lista única de caracteres do algoritmo. A Figura 17 mostra o código em C++ dessas funções.

A Figura 18 apresenta a solução otimizada e paralelizada do código mostrado na Figura 17. A primeira otimização realizada foi substituir todo o código da função `is_in()` por apenas uma chamada a uma função nativa do C++ `std::find()`. Essa modificação resulta em ganhos de desempenho, tendo em vista que a função `std::find()` é bem otimizada.

A segunda otimização foi a inclusão da paralelização do primeiro laço na função `generate_limited_data()` com uso da diretiva `#pragma omp parallel for` do OpenMP. Após a realização de diversos testes com escalonadores diferentes, o escalonador escolhido para este caso foi o `dynamic` com *chunk* de tamanho 10. O escalonador `dynamic` apresenta um resultado superior ao escalonamento `static`, pois o tempo de computação de cada iteração deste laço é variável (o tempo para encontrar se um elemento está ou não presente em um vetor não é constante). Isso causa um desbalanceamento de carga, o qual é atenuado pelo escalonador `dynamic`.

A mesma análise com `Valgrind` foi realizada no código de decodificação de textos. O resultado gerado pelo `Valgrind` é ilustrado na Figura 19. Como é possível observar na

```
1   bool is_in(int n, std::vector<int> vnData) {
2       bool bResult = false;
3
4       for (int i = 0; i < vnData.size(); i++) {
5           if (n == vnData[i])
6             bResult = true;
7       }
8
9       return bResult;
10  }
11
12  std::vector<int> generate_limited_data(std::vector<int> vnData) {
13      std::vector<int>tmp(256);
14      tmp[0] = vnData[0];
15      int k = 1;
16
17      for (int i = 1; i<vnData.size(); i++) {
18          if (!is_in(vnData[i], tmp)) {
19              tmp[k]=vnData[i];
20              k++;
21          }
22      }
23
24      std::vector<int>vnLimited(k);
25      for (int i = 0; i < k; i++)
26        vnLimited[i] = tmp[i];
27
28      return vnLimited;
29  }
```

Figura 17 – Funções `generate_limited_data()` e `is_in()`.

Figura 19, o método mais custoso em termos de tempo de execução é o `decode_vector()`, o qual usa o operador `[]` para acessar os elementos do vetor que contém os dados codificados.

A Figura 20 mostra o código da função, onde é possível notar que o método possui vários laços aninhados, sendo possível paralelizá-los com OpenMP de diferentes formas.

A solução adotada foi incluir o `#pragma omp parallel for` no laço mais externo, iniciando-se assim a região paralela na linha 8.

O escalonamento usado foi o `dynamic`, devido à imprevisibilidade do tempo que cada *thread* vai demorar pra executar o código, afinal é feita uma grande quantidade de computação dentro dos 4 laços que formam a região. Os cálculos feitos servem pra localizar dados dentro de um vetor, portanto uma *thread* pode achar o conteúdo logo no primeiro valor e outra *thread* achar o conteúdo no último valor, causando um desbalanceamento de carga, portanto o escalonamento dinâmico foi usado para evitar esse problema.

## 4.2 Cripto-compressão de Imagens em Paralelo

O `Valgrind` foi novamente utilizado para medir o custo de execução de cada função da aplicação do código de cripto-compressão de imagens. O resultado da análise está ilustrado na Figura 21.

```
1   bool is_in(int n, std::vector<int> vnData) {
2      return std::find(vnData.begin(), vnData.end(), n) != vnData.end();
3   }
4
5   std::vector<int> generate_limited_data(std::vector<int> vnData) {
6       std::vector<int>tmp(256);
7       tmp[0] = vnData[0];
8       int k = 1;
9
10      #pragma omp parallel for schedule(dynamic, 10)
11      for (int i = 1; i<vnData.size(); i++) {
12          if (!is_in(vnData[i], tmp)) {
13              tmp[k]=vnData[i];
14              k++;
15          }
16      }
17
18      std::vector<int>vnLimited(k);
19      for (int i = 0; i < k; i++)
20        vnLimited[i] = tmp[i];
21
22      return vnLimited;
23  }
```

Figura 18 – Versão otimizada e paralela das funções `generate_limited_data()` e `is_in()`.

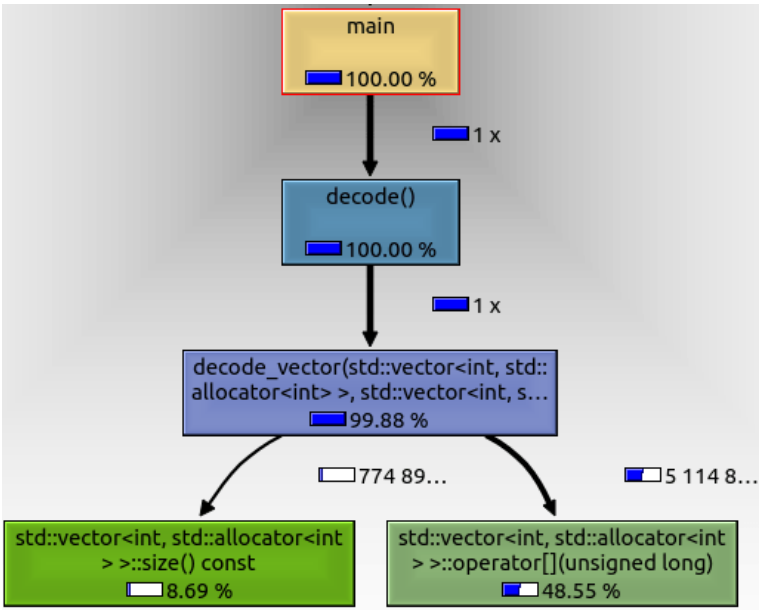



Figura 19 – Percentual do tempo de execução despendido nas principais funções do algoritmo original de decodificação de textos.

Na Figura 21 é possível perceber que, na parte de codificação, o método que representa uma grande parcela do tempo de execução é o `vec2compactstring`, além do próprio método `encode`, o qual apresenta muitos laços em seu código. O código da função `vec2compactstring` é exibido na Figura 22.

O código mostrado na Figura 22 foi otimizado com auxílio de um método nativo do C++ denominado `std::copy`. O código resultante é mostrado na Figura 23. Além de simplificado, o código resultante apresentou ganhos de desempenho.

```
1   std::vector<int> decode_vector(std::vector<int> cdata,
2   std::vector<int> nLimited, std::vector<int> K) {
3     std::vector<int> nDecoded;
4     unsigned int nLastIndex = cdata.size();
5     unsigned int nLSize = nLimited.size();
6     nDecoded.resize(3*nLastIndex);
7
8     #pragma omp parallel for schedule(dynamic, 10)
9     for (unsigned int r = 0; r < nLastIndex; ++r) {
10      int cdatar = cdata[r];
11
12      for (unsigned int i = 0; i < nLSize; ++i) {
13        int nLimited_i = K[0] * nLimited[i];
14
15        for (unsigned int j = 0; j < nLSize; ++j) {
16          int nLimited_j = K[1] * nLimited[j];
17
18          for (unsigned int k = 0; k < nLSize; ++k) {
19              ...
20          }
21        }
22      }
23    }
24
25    return nDecoded;
26  }
```

Figura 20 – Versão paralela da função `decode_vector()`.

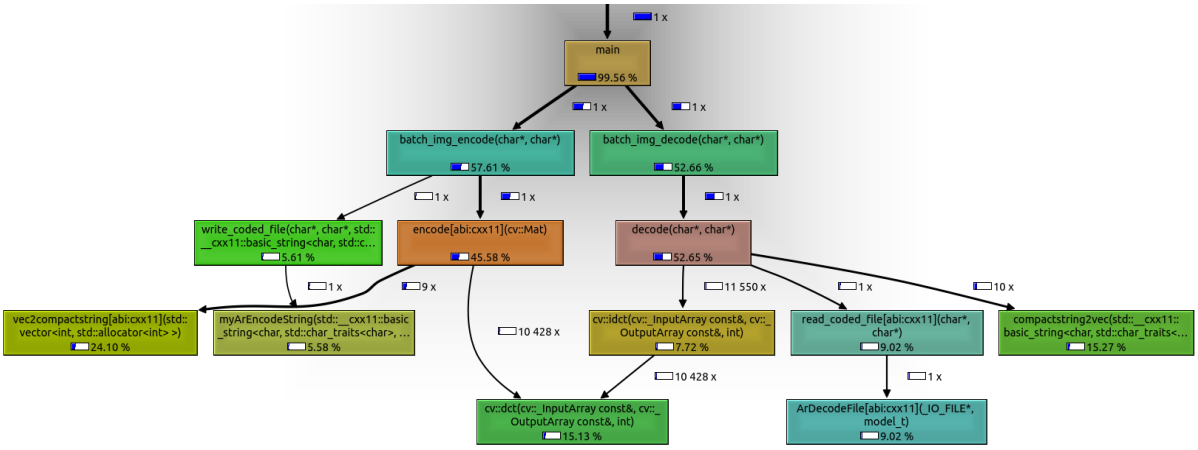



Figura 21 – Percentual do tempo de execução despendido nas principais funções do algoritmo original de codificação de imagens.

A paralelização com OpenMP foi realizada dentro dos métodos `encode` e `decode`. Um método, denominado `generate_quantization_matrix()`, o qual é utilizado tanto pelo `encode` quanto pelo `decode`, foi paralelizado também, conforme indicado na Figura 24. Este método é responsável por gerar a matriz que dividirá a imagem em blocos. A matriz gerada irá dividir a matriz da imagem original, gerando um conjunto de matrizes, que representam os diversos blocos. A linha 4 é responsável por iniciar a região paralela, na qual foi usado o escalonamento `guided`, devido à imprevisibilidade do tempo de execução por cada *thread*.

A função `encode` teve dois laços paralelizados, conforme indicado pela Figura 25. O

```
1   std::string vec2compactstring(std::vector<int> vec) {
2       std::string s;
3       for (int i=0; i<vec.size(); i++)
4       {
5           int a=vec.at(i);
6           std::stringstream ss;
7           ss << a;
8           std::string str = ss.str();
9           s+=str;
10      }
11      return s;
12  }
```

Figura 22 – Código original da função `vec2compactstring`.

```
1   std::string vec2compactstring(std::vector<int> vec) {
2       std::ostringstream vts;
3       std::copy(vec.begin(), vec.end(), std::ostream_iterator<int>(vts));
4       return vts.str();
5   }
```

Figura 23 – Código otimizado da função `vec2compactstring`.

```
1       cv::Mat generate_quantization_matrix(int Blocksize, int R) {
2         cv::Mat Q(Blocksize, Blocksize, CV_32F, cv::Scalar::all(0));
3
4         #pragma omp parallel for schedule(guided)
5         for (int r = 0; r < Blocksize; r++) {
6           for (int c = 0; c < Blocksize; c++) {
7             Q.at<float>(r,c) = (r == c == 0) ? 1 : (float)(r+c)*R;
8           }
9         }
10
11        return Q;
12      }
```

Figura 24 – Código da função `generate_quantization_matrix`.

primeiro é responsável por iterar sobre o vetor `planes`, que representa a imagem em blocos na forma de vetor, aplicando a DCT em cada bloco, ou elemento do vetor. O segundo é responsável por iterar sobre o mesmo vetor, porém removendo todos os zeros através da função `cv::findNonZero()` da biblioteca OPENCV. Também realiza a conversão do vetor para *string*.

Os laços utilizam o escalonamento `guided`, por conta da grande quantidade de computação que é feita dentro dos mesmos, sendo que as *threads* podem apresentar uma diferença de desempenho bastante grande, causando um desbalanceamento de carga. Logo, a estratégia `guided` se mostrou adequada para resolver esse problema. Ela foi escolhida no lugar da `dynamic`, pois de acordo com os testes realizados ela permitiu reduzir o sobrecusto de sincronizações e, ainda assim, garantir um bom balanceamento de carga.

A função `decode` teve três laços paralelizados, conforme indicado pela Figura 26. O primeiro é responsável pela leitura do arquivo codificado e por converter o mesmo para um vetor. O segundo insere no vetor todos os zeros que foram removidos na etapa de

```
1       std::string encode(cv::Mat matImg) {
2         ...
3
4         #pragma omp parallel for schedule(guided)
5         for (int i = 0; i < planes.size(); i++) {
6           ...
7           cv::dct(roi, dst);
8           ...
9         }
10
11        ...
12
13        #pragma omp parallel for schedule(guided)
14        for (int i = 0; i < planesSize; i++) {
15          ...
16          cv::findNonZero(matBoolean, matThreeLocXY[i]);
17          ...
18          szCompactThreeData[i] = vec2compactstring(vnThreeData[i]);
19          ...
20        }
21
22        ...
23      }
```

Figura 25 – Código da Função `encode`

codificação, com base no vetor que indica as posições corretas. O último realiza a função inversa da DCT, para recuperar o conteúdo original. O escalonamento escolhido também foi o `guided`, pelo mesmo motivo apresentado na parte de codificação.

```
1       cv::Mat decode(char* filePathIn, char* fileName) {
2           ...
3
4           #pragma omp parallel for schedule(guided)
5           for (int i = 0; i < planes; i++) {
6               ...
7           }
8
9           #pragma omp parallel for schedule(guided)
10          for (int i = 0; i < planes; i++) {
11              ...
12              int index0 = 0;
13              for (int j = 0; j < vnDiffZeroLoc[i].size(); j++) {
14                index0 += vnDiffZeroLoc[i][j];
15                vec[index0] = 0;
16              }
17
18              int index = 0;
19              for (int j = 0; j < vec.size(); j++) {
20                if ( vec[j] == -1) {
21                  vec[j] = vnNonZeroData[i][index];
22                  index++;
23                }
24              }
25              ...
26          }
27
28          #pragma omp parallel for schedule(guided)
29          for (int i = 0; i < planes; i++) {
30              ...
31              cv::idct(roi2, dst);
32              ...
33          }
34
35          ...
36      }
```

Figura 26 – Código da Função `decode`

# 5 Resultados

Este capítulo apresenta a plataforma utilizada para os testes, todos os resultados obtidos nos testes realizados, seus gráficos e uma análise dos mesmos.

## 5.1 Ambiente Experimental

A plataforma utilizada para os experimentos foi a máquina `Tesla`, do Laboratório de Pesquisa em Sistemas Distribuídos (LaPeSD) da UFSC. A máquina é equipada com dois CPUs `Intel(R) Xeon(R) E5-2640`, que possuem 10 núcleos físicos e mais 10 núcleos lógicos cada, com a tecnologia `Hyper-Threading`. A frequência de operação do processador é de 2,40 GHz. A máquina possui 128 GB de memória RAM, além de uma GPU `NVIDIA Tesla K40c`. A arquitetura de memória que a máquina possui é a NUMA, conforme visto na Seção 2.1.1.1. O *software* da `Tesla` é composto pelo `Ubuntu 16.04.4 LTS`, em conjunto com o *kernel* do Linux em sua versão 4.4.0.

Como a máquina utilizada nos experimentos possui uma arquitetura de memória NUMA, foram estudadas três maneiras de escalonar as *threads* (*thread pinning*) das regiões paralelas do código nos núcleos de processamento. Por padrão, o OpenMP deixa à cargo do Sistema Operacional decidir em qual núcleo cada *thread* deve ser executada. Além disso, o próprio Sistema Operacional pode decidir por realizar migrações de *threads* durante a execução da aplicação. Esta estratégia será representada nos gráficos como "padrão". Além do escalonamento de *threads* padrão, foram testadas duas estratégias que tentam tirar proveito da arquitetura NUMA. A primeira delas, denominada *compact*, fixa as *threads* nos núcleos, ocupando primeiramente todos os núcleos físicos de uma CPU para, então, utilizar os núcleos físicos de outra CPU. A segunda delas, denominada *scatter*, tenta balancear as *threads* entre entre as CPUs. Para isso, ela realiza a distribuição de *threads* entre CPUs de acordo com um algoritmo *round-robin*. Por exemplo, em uma execução com 5 *threads* em uma máquina com 2 CPUs, as *threads* 0, 2, 4 seriam fixadas na CPU 0 e as *threads* 1, 3 na CPU 1. Ambas as estratégias, *compact* e *scatter*, são técnicas já existentes e foram apenas utilizadas neste trabalho.

Além das estratégias de *thread pinning*, foram considerados diferentes tamanhos de arquivos texto para codificação (1MB, 2MB, 4MB e 8MB), diferentes tamanhos de texto para decodificação (100KB, 200KB, 400KB e 800KB), diferentes resoluções de imagens (360p, 720p, 1080p e 4K) e diferentes números de *threads* (de 1 até 10). Os tamanhos dos textos usados para codificação e decodificação foram diferentes devido à parte de decodificação precisar de um tempo consideravelmente maior que a parte de codificação para ser executada. Os tamanhos escolhidos para as imagens de teste foram selecionados

com base nos tamanhos de maior utilização por parte dos usuários de computadores, sendo essas as dimensões padrão de muitos equipamentos, como smartphones e TVs por exemplo.

As métricas de desempenho consideradas foram o tempo de execução (medido em segundos) e o *speedup*, o qual é calculado dividindo-se o tempo da versão sequencial do programa pelo tempo da versão paralela com $n$ threads. Cada experimento foi repetido 10 vezes e os resultados apresentados nos gráficos representam a média aritmética dos experimentos. No geral, o desvio padrão máximo observado foi de 0,19.

Para cada versão do algoritmo de cripto-compressão (texto e imagem) foram realizadas as seguintes análises:

- Análise de desempenho (*speedup*) dos métodos de codificação e decodificação variando-se o número de *threads* e a estratégia de *thread pinning*;
- Análise de desempenho (*speedup*) dos métodos de codificação e decodificação variando-se o número de *threads* e o tamanho do arquivo de entrada;
- Análise do tempo de execução total da aplicação, comparando-se a versão serial original do algoritmo GMPR, a versão serial otimizada em C++ e a versão paralela (a qual inclui as otimizações da versão serial C++).

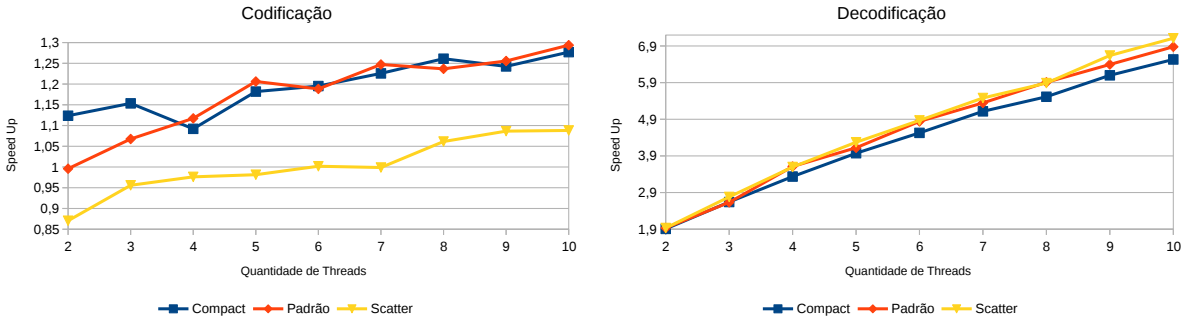
## 5.2 Cripto-compressão de Texto em Paralelo

### 5.2.1 Thread Pinning

Primeiramente foram realizados experimentos para avaliar o impacto da estratégia de *thread pinning* no desempenho da codificação e decodificação em paralelo. Para a realização destes testes foi utilizado um arquivo de texto de 8MB para codificação e outro de 800KB para decodificação. A Figura 27a mostra a comparação entre as 3 estratégias de *thread pinning* para a codificação de textos.

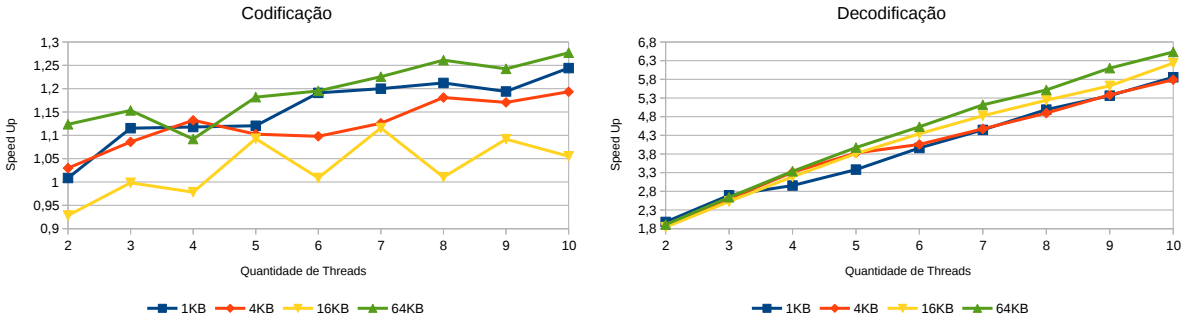
Conforme é possível perceber, as estratégias sem fixar threads e a `Compact` apresentaram, de forma similar, os melhores resultados, sendo a `Compact` levemente superior, portanto ela será utilizada na parte de codificação de textos nos demais experimentos apresentados neste trabalho. A estratégia `Compact` apresentou desempenho superior pois a arquitetura do ambiente experimental é a NUMA, onde o tempo de acesso à memória é diferente para cada processador, sendo a estratégia que aloca todas as threads no mesmo processador a mais eficiente, reduzindo a latência.

A Figura 27b mostra a comparação entre as 3 estratégias de *thread pinning* para a decodificação de textos. Neste caso a melhor estratégia foi a `Scatter`, conforme pode ser




(a) Codificação (b) Decodificação

Figura 27 – Impacto da estratégia de *thread pinning* no desempenho da versão paralela da cripto-compressão de textos.




(a) Codificação (b) Decodificação

Figura 28 – Impacto do tamanho dos arquivos no desempenho da versão paralela da cripto-compressão de textos.

visto no gráfico, sendo portanto a escolhida para a parte de decodificação do algoritmo de textos para os demais experimentos apresentados neste trabalho.

### 5.2.2 Tamanhos de Texto

Os testes compararam textos de tamanho 1MB, 2MB, 4MB e 8MB para a parte de codificação e textos de tamanho 100KB, 200KB, 400KB e 800KB para a parte de decodificação. As estratégias de *thread pinning* adotadas foram a `Compact`, para a codificação, e a `Scatter`, para a decodificação.

A Figura 28a mostra a comparação entre o tamanho dos textos e a quantidade de *threads* utilizadas para a codificação. Conforme exibido no gráfico, todos os diferentes textos mostraram um desempenho similar. Observou-se um pequeno ganho de desempenho com o aumento do número de *threads* utilizadas na computação. O *speedup* máximo obtido foi baixo (inferior a 1,3x), o que indica um ganho não muito significativo com o paralelismo.

A Figura 28b mostra a comparação entre o tamanho dos textos e a quantidade de *threads* utilizadas para a decodificação. Neste gráfico é possível perceber o `speedup` crescente

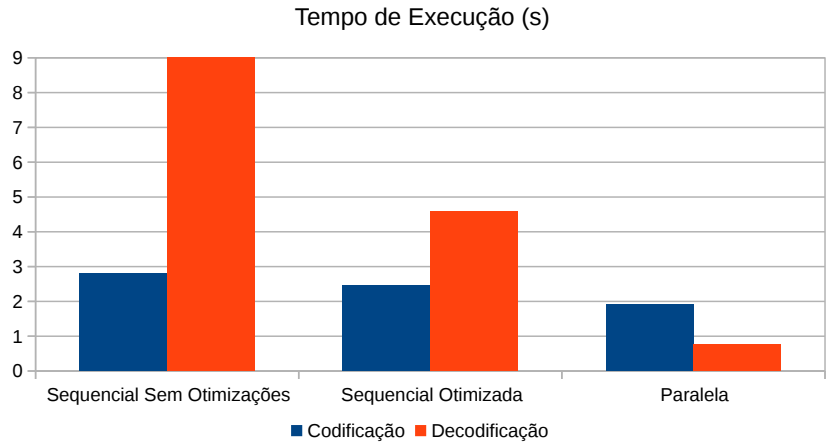



Figura 29 – Tempo de execução total do GMPR para textos.

de acordo com a quantidade de *threads*, mostrando que o paralelismo foi responsável por um ganho considerável de desempenho. O mesmo grau de performance foi conseguido em todos os diferentes textos.

### 5.2.3 Tempo de Execução

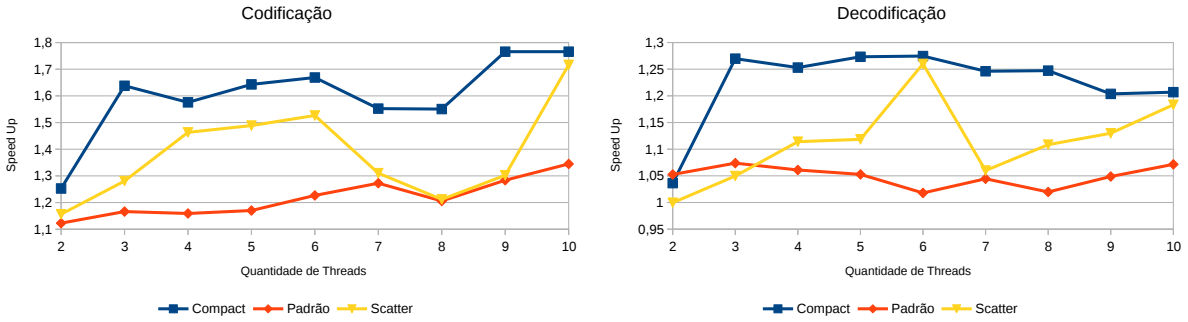
Este teste comparou o algoritmo GMPR usando um texto de tamanho 8MB para a codificação e outro de tamanho 200KB para a decodificação. Foi comparada a versão original do código em C, a versão serial otimizada em C++ proposta neste TCC e a versão paralelizada proposta neste TCC. Nestes experimentos, foram utilizadas 10 *threads* e as técnicas `Compact`, para codificação, e `Scatter`, para decodificação. A Figura 29 exibe os tempos absolutos de execução do algoritmo em segundos (s).

Como se pode perceber no gráfico, a parte de codificação obteve um ganho pequeno de desempenho, por conter pouca quantidade de computação envolvida. Por outro lado, a parte de decodificação, que contém uma quantidade maior de computação, apresentou um ganho significativo de desempenho, tanto pela otimização sequencial quanto pelo paralelismo, chegando a reduzir o tempo de execução em 11,9x, conforme mostra o gráfico. As grandes diferenças nos tempos de execução se devem ao fato de que a parte de decodificação possui muitos laços possíveis de paralelização, enquanto a codificação não possui a mesma quantidade de processamento.

## 5.3 Cripto-compressão de Imagens em Paralelo

### 5.3.1 Thread Pinning

Os testes de *thread pinning* usaram uma imagem de tamanho 4K, tanto para codificação quanto para decodificação. A Figura 30a mostra a comparação entre as 3




(a) Codificação (b) Decodificação

Figura 30 – Impacto da estratégia de *thread pinning* no desempenho da versão paralela da cripto-compressão de imagens.

estratégias de *thread pinning* para a codificação de imagens. Conforme é possível perceber, a estratégia que apresenta o melhor desempenho é a `Compact`, a qual fixa todas as *threads* na mesma CPU, distribuindo em ordem crescente entre os núcleos disponíveis.

A Figura 30b mostra a comparação entre as 3 estratégias de *thread pinning* para a decodificação de imagens. Conforme o gráfico mostra, a estratégia `Compact` novamente apresenta um desempenho melhor, sendo portanto a escolhida para a versão de imagens do algoritmo GMPR.

A estratégia `Compact` apresentou os melhores resultados pelos mesmos motivos citados na versão de texto do algoritmo, reduzindo a latência no acesso à memória, devido à arquitetura NUMA da máquina experimental.

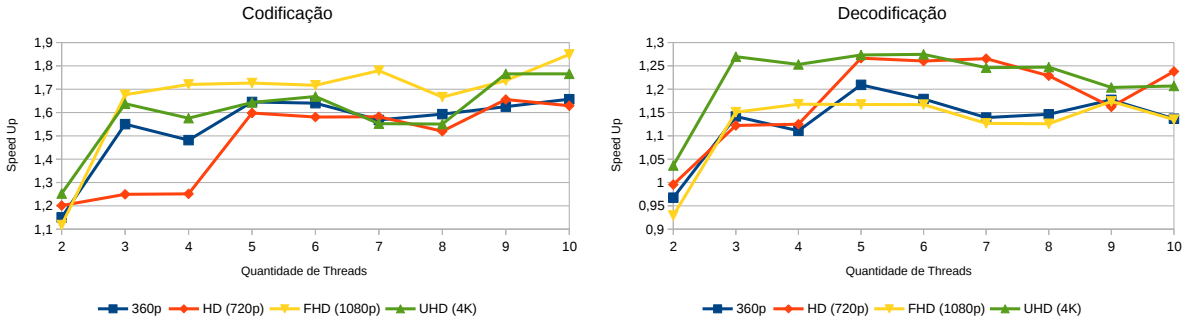
### 5.3.2 Tamanhos de Imagem

Os testes compararam imagens de tamanho 360p, HD (720p), Full HD (1080p) e Ultra HD (4K). A estratégia de *thread pinning* adotada foi a `Compact`, que apresentou os melhores resultados no teste anterior. A Figura 31a mostra a comparação entre o tamanho das imagens e a quantidade de *threads* utilizadas para a codificação de imagens.

Conforme exibido no gráfico, todos os tamanhos de imagem obtiveram um `speedup` crescente em relação ao número de *threads* utilizadas. O `speedup` tende a ser maior para imagens maiores, como é possível perceber com base na Figura 31a.

A Figura 31b mostra a comparação entre o tamanho das imagens e a quantidade de *threads* utilizadas para a decodificação de imagens. Este gráfico mostra um desempenho bastante similar entre as diferentes imagens, sendo que todas obtiveram um `speedup` praticamente constante em relação à quantidade de *threads*.

Tanto no teste de *thread pinning* quanto no teste de tamanho das imagens, pode-se perceber que a parte de codificação apresenta uma maior variação no seu `speedup` em relação a quantidade de *threads* usadas, enquanto a parte de decodificação não apresenta




(a) Codificação (b) Decodificação

Figura 31 – Impacto do tamanho dos arquivos no desempenho da versão paralela da cripto-compressão de imagens.

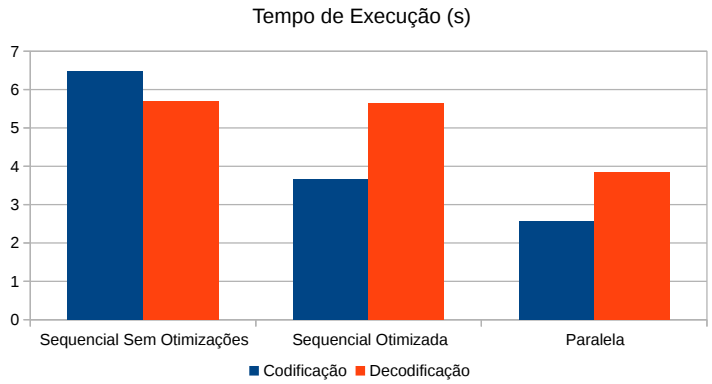



Figura 32 – Tempo de Execução do GMPR para Imagens

uma diferença significativa. Isso se deve ao fato de que a parte de decodificação possui a sua maior parte da computação sendo executada de forma serial, sendo possível aumentar a performance com poucas *threads*, mas não tendo muitos ganhos ao aumentar muito a quantidade das mesmas.

### 5.3.3 Tempo de Execução

Este teste comparou o algoritmo GMPR usando uma imagem de tamanho Ultra HD (4K). Foi comparada a versão original do código em C, a versão serial otimizada em C++ por este TCC e a versão paralelizada proposta neste TCC. Os experimentos foram executados utilizando-se 10 *threads* e a técnica `Compact` de *thread pinning*.

A Figura 32 exibe os tempos absolutos de execução do algoritmo em segundos (s).

Como se pode perceber no gráfico, a parte de codificação ganhou bastante desempenho na otimização sequencial e também no uso de paralelismo, enquanto a parte de decodificação apresentou um ganho mais tímido, obtendo uma performance melhor na versão paralela. O ganho mais tímido observado na parte de decodificação, na otimização sequencial, se deve ao fato de que não foi possível utilizar funções nativas da linguagem, ao

passo que isso foi feito na parte de codificação, explicando parte das diferenças encontradas nos tempos de execução.

# 6 Conclusão

O algoritmo GMPR funciona corretamente, cumprindo seu objetivo com maestria, porém apresenta um desempenho insatisfatório, principalmente na versão de textos. Este trabalho buscou analisar o código e melhorá-lo, garantindo que seu tempo de execução diminua e o torne possível de ser utilizado em aplicações reais.

A utilização da API OpenMP foi a solução escolhida para melhorar o desempenho do algoritmo de cripto-compressão GMPR, em conjunto com outras otimizações de código que não envolveram paralelismo. A implementação sequencial otimizada e a paralela tiveram seus resultados medidos e foram melhoradas gradativamente, de forma a garantir um ganho de performance.

Ter o algoritmo funcionando de forma eficiente é uma conquista muito interessante, pois o mesmo poderá ser utilizado nos aeroportos do Reino Unido sem se tornar um gargalo tanto em tempo de execução quanto no tamanho dos arquivos de imagem, reduzindo o armazenamento utilizado, os riscos de segurança e também o tempo necessário para enviar informações através da Internet.

O bom funcionamento do algoritmo também é interessante pois futuramente será expandido para cripto-comprimir qualquer tipo de conteúdo, sendo uma solução muito boa tanto para a academia, quanto para empresas ou usuários comuns.

Os resultados obtidos neste trabalho foram significantes, obtendo uma boa redução no tempo necessário para a execução do algoritmo GMPR. As execuções foram 1,47x mais rápidas na codificação e 11,9x na decodificação para a parte de textos, e 2,53x mais rápidas na codificação e 1,48x na decodificação para a parte de imagens.

Possíveis melhorias ou trabalhos futuros neste algoritmo poderiam envolver paralelismo em múltiplos computadores, através do MPI, ou a execução em GPUs, através do CUDA ou OPENCL. Também pode ser estudado um ganho de desempenho ao se usar uma implementação paralela da biblioteca usada no algoritmo de imagens, a OPENCV, assim como versões paralelas das funções nativas do C++, contidas na biblioteca padrão da linguagem.

O uso deste trabalho em outros projetos também é bastante possível, afinal esse algoritmo pode ser adaptado para o uso em projetos que tenham restrições de segurança ou armazenamento. Há também a possibilidade desse projeto ser estendido para outras áreas, como a cripto-compressão de um banco de dados, dentre outras.

O código fonte do projeto em C++ está disponível no GitHub[1].

[1] <https://github.com/leandroperin/ParallelCryptoCompression>

# Referências

CHANDRA, R. et al. *Parallel Programming in OpenMP*. [S.l.]: Morgan Kaufmann, 2000. 231 p. ISBN 978-1558606715. Citado na página 24.

CHAPMAN, B.; JOST, G.; PAS, R. *Using OpenMP: Portable Shared Memory Parallel Programming*. [S.l.]: MIT Press, 2008. 353 p. ISBN 978-0262533027. Citado 2 vezes nas páginas 21 e 25.

FLYNN, M. J. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, IEEE, v. 21, n. 9, p. 948–960, September 1972. Disponível em: <http://ieeexplore.ieee.org/document/5009071/>. Citado na página 17.

HWANG, K.; XU, Z. *Scalable Parallel Computing: Technology, Architecture, Programming*. First. [S.l.]: McGraw-Hill Science/Engineering/Math, 1998. 832 p. ISBN 978-0070317987. Citado na página 20.

MCCOOL, M.; REINDERS, J.; ROBISON, A. D. *Structured Parallel Programming: Patterns for Efficient Computation*. [S.l.]: Morgan Kaufmann, 2012. 406 p. ISBN 978-0124159938. Citado na página 21.

MIRANDA, C. S. A evolução da gpgpu: arquitetura e programação. p. 1–7, 2010. Disponível em: <http://www.ic.unicamp.br/~ducatte/mo401/1s2010/T2/070498-t2.pdf>. Citado na página 20.

PFLEEGER, C. P.; PFLEEGER, S. L.; MARGULIES, J. *Security in Computing*. [S.l.]: Prentice Hall, 2015. 910 p. ISBN 978-0134085043. Citado na página 27.

RODRIGUES, M. A.; SIDDEQ, M. M. Information systems: Secure access and storage in the age of cloud computing. *Athens Journal of Sciences*, Athens Institute for Education and Research, v. 3, n. 4, p. 267–284, September 2016. Disponível em: <http://shura.shu.ac.uk/13715/>. Citado 4 vezes nas páginas 13, 14, 33 e 35.

SALOMON, D. *Data Compression: The Complete Reference*. [S.l.]: Springer Science & Business Media, 2007. 1092 p. ISBN 978-1846286032. Citado 2 vezes nas páginas 13 e 29.

STALLINGS, W. *Cryptography and Network Security: Principles and Practice*. [S.l.]: Pearson, 2014. 731 p. ISBN 978-0133354690. Citado 3 vezes nas páginas 13, 25 e 26.

TANENBAUM, A. S.; BOS, H. *Modern Operating Systems*. Fourth. [S.l.]: Pearson, 2015. 1137 p. ISBN 978-0-13-359162-0. Citado 2 vezes nas páginas 17 e 20.

# Apêndices

# APÊNDICE A – Artigo

## Uso de Computação Paralela Para Acelerar a Cripto-Compressão de Dados

**Leandro P. Oliveira**[1]

[1] Departamento de Informática e Estatística
Universidade Federal de Santa Catarina (UFSC) – Florianópolis, SC – Brazil

leandro.perino@gmail.com

***Abstract.*** *In this paper we will propose performance optimizations for the data crypto-compression algorithm called GMPR, which was developed at Sheffield Hallam University in order to compress and encrypt 3D images in airport security systems. Two simplified versions of the algorithm were considered: One for texts and one for images. The optimizations were based on code rewriting, use of native C++ functions and parallelism.*

***Resumo.*** *Neste trabalho serão propostas otimizações de desempenho para o algoritmo de cripto-compressão de dados denominado GMPR, o qual foi desenvolvido na Sheffield Hallam University com o objetivo de comprimir e cifrar imagens 3D em sistemas de segurança aeroportuária. Foram consideradas duas versões simplificadas do algoritmo: Uma para textos e outra para imagens. As otimizações se basearam em reescrita do código, uso de funções nativas do C++ e paralelismo.*

## 1. Introdução

Serviços como o YouTube, Instagram, Facebook, dentre outros, possuem uma enorme quantidade de dados armazenados. Com o aumento no uso de serviços como esses, maior tráfego de dados através da rede e necessidade de armazenamento cada vez maiores, são requeridos métodos mais eficientes para compressão de imagens e vídeos, com alta qualidade de reconstrução e redução na quantidade de armazenamento necessária para os dados.

Portanto, a disponibilidade de um algoritmo de compressão e cifragem de dados mais eficiente, que consiga reduzir mais o tamanho dos arquivos, comprimir e criptografar de forma rápida, e ainda assim mantendo um bom desempenho na recuperação dos dados é algo que melhoraria bastante o uso de serviços com alto tráfego de informações. Tal algoritmo tornaria os serviços em nuvem mais populares, afinal deixaria os mesmos mais rápidos e seguros, encorajando o desenvolvimento de cada vez mais aplicativos que fazem uso de muitos dados.

Um exemplo de um sistema que necessita de algoritmos eficientes para criptografia e compressão de dados pode ser encontrado em aeroportos do Reino Unido. Um sistema de reconhecimento de rostos em 3D foi desenvolvido, o problema é que o mesmo estava gerando uma quantia significativa de dados, aproximadamente 20MB para cada rosto em 3D, sendo duas capturas para cada passageiro, uma no check-in e outra no portão de

embarque. Para exemplificar, um voo com 400 passageiros iria gerar 8GB de dados, que precisariam ser enviados para a Polícia Metropolitana de Londres e para a polícia do local de destino, e num aeroporto que tem um fluxo de aproximadamente 200 mil passageiros por dia, gerar 4TB de dados diariamente causaria um grande problema para trocar informações com celeridade.

Pesquisadores da Sheffield Hallam University desenvolveram um algoritmo de cripto-compressão, chamado GMPR, para resolver o problema de enviar imagens 3D com grande quantidade de informações de forma segura. O algoritmo de compressão desenvolvido apresentou resultados impressionantes em contraste com os formatos 3D tradicionais. Assim surgiu a ideia de estender o código para comprimir imagens, texto e outros formatos de dados.

Eventualmente foi pensado que o mesmo poderia ser utilizado em aspectos de segurança, como uma forma de criptografia. A conclusão foi de que apenas o cabeçalho dos arquivos seriam criptografados com o conhecido algoritmo AES, sendo o corpo dos arquivos apenas comprimidos, sendo impossível visualizar os dados corretamente por conta do cabeçalho estar cifrado com o AES.

Embora os algoritmos de cripto-compressão mencionados funcionem corretamente, eles ainda demoram um tempo considerável para processar conteúdos grandes, como imagens de alta resolução ou textos com milhares de linhas, o que torna inviável o uso dos mesmos em aplicações que exigem resposta rápida aos usuários. Este trabalho irá propor uma versão mais simplificada e eficiente do GMPR, focada em textos e imagens 2D.

## 2. O Algoritmo GMPR

O algoritmo GMPR possui duas implementações distintas, uma com foco em cripto-compressão de texto e outra em imagens, sendo que ambas as versões diferem no método de compressão, porém funcionam da mesma maneira para a criptografia.

### 2.1. Criptografia

São geradas 3 chaves na etapa de compressão que são necessárias para recuperar o conteúdo original do arquivo. As chaves são geradas da seguinte forma:

K[0]←rand() % 1001

K[1]←(rand() % 11) * 2 *maxvalue

K[2]←K[1]*maxvalue* (rand() % 11)

Onde o *maxvalue* equivale ao maior caractere do texto plano. Essas chaves são armazenadas no cabeçalho dos arquivos.

Posteriormente, o cabeçalho é cifrado com o conhecido algoritmo AES, garantindo a segurança do mesmo. O corpo do arquivo não precisa ser cifrado, pois é necessário o conteúdo do cabeçalho para recuperar o conteúdo do corpo, logo o mesmo fica apenas comprimido, sem a necessidade de outra camada de segurança.

### 2.2. Compressão de Textos

De modo geral, o algoritmo GMPR é iniciado obtendo o texto plano e convertendo-o

para ASCII. Em seguida é obtida uma lista dos caracteres contidos no arquivo, sem repetir nenhum.

Então são geradas as 3 chaves aleatórias. A partir disso é gerada uma lista chamada *nCoded*, na qual a cada 3 caracteres do texto é gerado 1 elemento codificado usando as chaves geradas O primeiro caractere é multiplicado pela primeira chave, o segundo caractere pela segunda chave e o terceiro caractere pela terceira chave, então os 3 valores são somados e adicionados na lista *nCoded*, prosseguindo com os próximos 3 caracteres.

É criado um vetor chamado *codedvector* que é formado pelas 3 chaves, pela lista única de caracteres e pela lista *nCoded*. Em seguida o texto final é gerado. A taxa de compressão é de 1/3 do tamanho original, afinal 1 elemento do texto comprimido representa 3 elementos do texto original.

### 2.3. Descompressão de Textos

Para descomprimir o conteúdo e obter o arquivo original, o algoritmo começa obtendo o texto cifrado e transformando-o em um vetor. Em seguida são extraídas as 3 chaves usadas no processo de cifragem, a lista única de caracteres e a lista *nCoded*. Por fim, o conteúdo é recuperado através da seguinte verificação:

```
if data[r] ==K[0]*nLimited[i] +K[1]*nLimited[j] +K[2]*nLimited[k] then
```

Onde o vetor *data* representa o texto codificado e o vetor *nLimited* representa a lista única de caracteres.

### 2.4. Compressão de Imagens

Primeiramente a imagem é dividida em blocos de tamanho n, e então é aplicada a Transformada Discreta do Cosseno (DCT) em cada bloco.

Em seguida é aplicado o algoritmo de minimização de matrizes nos blocos, eliminando todos os zeros, que geralmente são muitos, reduzindo seu tamanho e produzindo o vetor minimizado. São definidas as 3 chaves aleatórias que irão multiplicar cada 3 entradas deste vetor minimizado, e então os 3 valores são somados, ou seja, cada 3 valores do vetor é transformado em apenas 1, reduzindo o tamanho para 1/3 do original, produzindo o vetor minimizado codificado.

### 2.5. Descompressão de Imagens

A etapa de descompressão inicia na leitura do arquivo codificado e converte o mesmo para um vetor. Em seguida são recuperadas as 3 chaves, o vetor sem zeros que representa a imagem, a posição dos zeros removidos e a lista única de caracteres.

É feita a busca binária no vetor sem zeros, combinando as 3 chaves e recuperando assim o conteúdo completo. Em seguida os zeros são colocados novamente nas posições indicadas pelo vetor de posições.

Por último, é realizada a operação inversa da DCT e a imagem original é recuperada e salva em disco.

## 3. Paralelização da Cripto-Compressão

Este trabalho possui três etapas principais. Na primeira etapa, foi feita uma

implementação em C++ do algoritmo de cripto-compressão GMPR, incluindo-se algumas otimizações quando possível. Com base nesta solução proposta, na segunda etapa foram estudadas formas de paralelizar o código para obter-se um desempenho ainda melhor. A API do OpenMP foi explorada em detalhes para implementar soluções paralelas eficientes para este problema. Por fim, na terceira etapa, foram realizados experimentos para medir o desempenho das soluções paralelas propostas.

O objetivo inicial foi definir em quais partes do código o programa leva mais tempo para executar, analisar se é possível quebrar essa parte em pedaços pequenos e atribuir múltiplas threads para a execução. Para isso, alguns experimentos preliminares foram realizados utilizando-se a ferramenta Valgrind no Linux.

Após obter os resultados, o próximo passo foi transformar os setores mais custosos do código em regiões paralelas sempre que possível, a fim de melhorar o desempenho da aplicação. Também foi verificada a existência de condições de corrida no código paralelizado. Nesses casos, regiões críticas do código, que são regiões que só podem ser executadas em uma thread de cada vez, deverão ser protegidas com o uso da diretiva omp critical, que limita a execução pelas threads.

### 3.1. Cripto-compressão de Textos em Paralelo

A primeira otimização realizada foi substituir todo o código da função is_in() por apenas uma chamada a uma função nativa do C++, a std::find(). Essa modificação resulta em ganhos de desempenho, tendo em vista que a função std::find() é bem otimizada.

A segunda otimização foi a inclusão da paralelização do primeiro laço na função generate_limited_data() com uso da diretiva #pragma omp parallel for do OpenMP. Após a realização de diversos testes com escalonadores diferentes, o escalonador escolhido para este caso foi o dynamic com chunk de tamanho 10. O escalonador dynamic apresenta um resultado superior ao escalonamento static, pois o tempo de computação de cada iteração deste laço é variável, causando um desbalanceamento de carga, o que é atenuado pelo escalonador dynamic.

A terceira otimização foi a paralelização do laço mais externo na função decode_vector(), também utilizando a diretiva #pragma omp parallel for do OpenMP, com escalonador dynamic e chunk de tamanho 10.

### 3.2. Cripto-compressão de Imagens em Paralelo

O código foi otimizado com auxílio de um método nativo do C++ denominado std::copy. O código resultante, além de simplificado, apresentou ganhos de desempenho.

A paralelização com OpenMP foi realizada dentro dos métodos encode e decode. Um método, denominado generate_quantization_matrix(), o qual é utilizado tanto pelo encode quanto pelo decode, foi paralelizado também.

A função encode teve dois laços paralelizados. O primeiro é responsável por iterar sobre o vetor planes, que representa a imagem em blocos na forma de vetor, aplicando a DCT em cada bloco, ou elemento do vetor. O segundo é responsável por iterar sobre o mesmo vetor, porém removendo todos os zeros através da função cv::findNonZero() da

biblioteca OPENCV. Também realiza a conversão do vetor para string.

A função decode teve três laços paralelizados. O primeiro é responsável pela leitura do arquivo codificado e por converter o mesmo para um vetor. O segundo insere no vetor todos os zeros que foram removidos na etapa de codificação, com base no vetor que indica as posições corretas. O último realiza a função inversa da DCT, para recuperar o conteúdo original.

Os laços utilizam o escalonamento guided, por conta da grande quantidade de computação que é feita dentro dos mesmos. Ela foi escolhida no lugar da dynamic, pois de acordo com os testes realizados ela permitiu reduzir o sobrecusto de sincronizações e, ainda assim, garantir um bom balanceamento de carga.

## 4. Resultados

O primeiro parâmetro variado nos experimentos foi a estratégia de thread pinning. A primeira delas, conhecida por ser a padrão, é aquela que o sistema operacional fica responsável por alocar as threads nos núcleos disponíveis. A segunda, conhecida por compact, é aquela que fixa as threads nos primeiros núcleos de um processador. A terceira, chamada scatter, distribui as threads entre núcleos de 2 ou mais processadores.

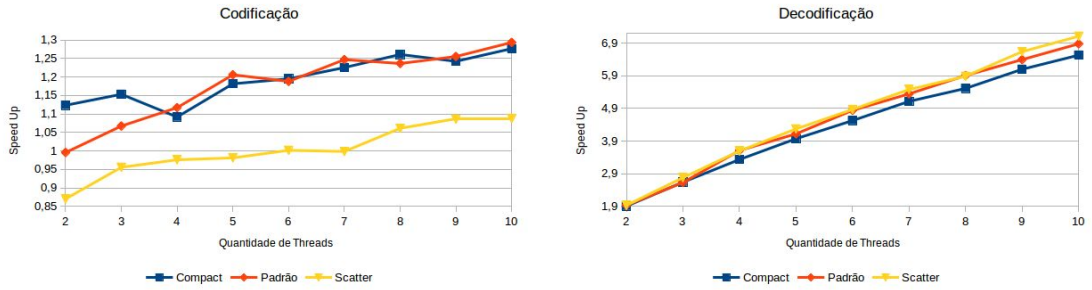
Além das estratégias de thread pinning, foram considerados diferentes tamanhos de arquivos texto para codificação (1MB, 2MB, 4MB e 8MB), diferentes tamanhos de texto para decodificação (100KB, 200KB, 400KB e 800KB), diferentes resoluções de imagens (360p, 720p, 1080p e 4K) e diferentes números de threads(de 1 até 10). Os tamanhos dos textos usados para codificação e decodificação foram diferentes devido à parte de decodificação precisar de um tempo consideravelmente maior que a parte de codificação para ser executada. Os tamanhos escolhidos para as imagens de teste foram selecionados com base nos tamanhos de maior utilização por parte dos usuários de computadores, sendo essas as dimensões padrão de muitos equipamentos, como smartphones e TVs por exemplo.

As métricas de desempenho consideradas foram o tempo de execução (medido em segundos) e o speedup, o qual é calculado dividindo-se o tempo da versão sequencial do programa pelo tempo da versão paralela com n threads. Cada experimento foi repetido 10 vezes e os resultados apresentados nos gráficos representam a média aritmética dos experimentos. No geral, o desvio padrão máximo observado foi de 0,19.

### 4.1. Cripto-compressão de Textos

Conforme é possível perceber na Figura 1a, as estratégias sem fixar threads e a Compact apresentaram, de forma similar, os melhores resultados, sendo a Compact levemente superior, portanto ela será utilizada na parte de codificação de textos nos demais experimentos apresentados neste trabalho. A estratégia Compact apresentou desempenho superior pois a arquitetura do ambiente experimental é a NUMA, onde o tempo de acesso à memória é diferente para cada processador, sendo a estratégia que aloca todas as threads no mesmo processador a mais eficiente, reduzindo a latência.

A Figura 1b exibe a parte de decodificação de textos. Neste caso a melhor estratégia foi a Scatter, portanto será a utilizada nos demais experimentos.

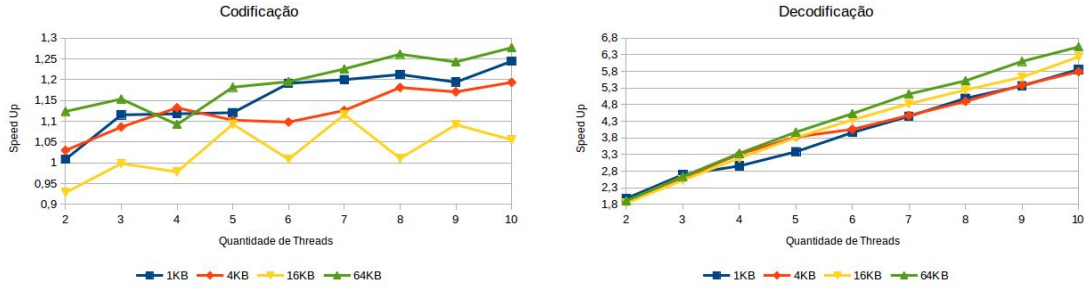



**(a) Codificação** **(b) Decodificação**

**Figura 1. Impacto das estratégias de thread pinning na cripto-compressão de textos**

A Figura 2a mostra a comparação entre o tamanho dos textos e a quantidade de threads utilizadas para a codificação. Conforme exibido no gráfico, todos os diferentes textos mostraram um desempenho similar. Observou-se um pequeno ganho de desempenho com o aumento do número de threads utilizadas na computação. O speedup máximo obtido foi baixo (inferior a 1,3x), o que indica um ganho não muito significativo com o paralelismo.

A Figura 2b mostra a comparação entre o tamanho dos textos e a quantidade de threads utilizadas para a decodificação. Neste gráfico é possível perceber o speedup crescente de acordo com a quantidade de threads, mostrando que o paralelismo foi responsável por um ganho considerável de desempenho. O mesmo grau de performance foi conseguido em todos os diferentes textos.




**(a) Codificação** **(b) Decodificação**

**Figura 2. Impacto do tamanho dos arquivos na cripto-compressão de textos**

Como se pode perceber no gráfico da Figura 3, a parte de codificação obteve um ganho pequeno de desempenho, por conter pouca quantidade de computação envolvida. Por outro lado,a parte de decodificação, que contém uma quantidade maior de computação, apresentou um ganho significativo de desempenho, tanto pela otimização sequencial quanto pelo paralelismo, chegando a reduzir o tempo de execução em 11,9x, conforme mostra o gráfico.

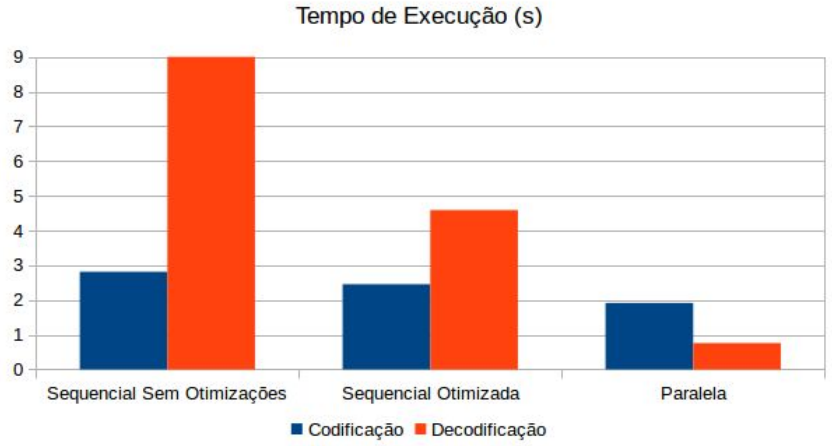


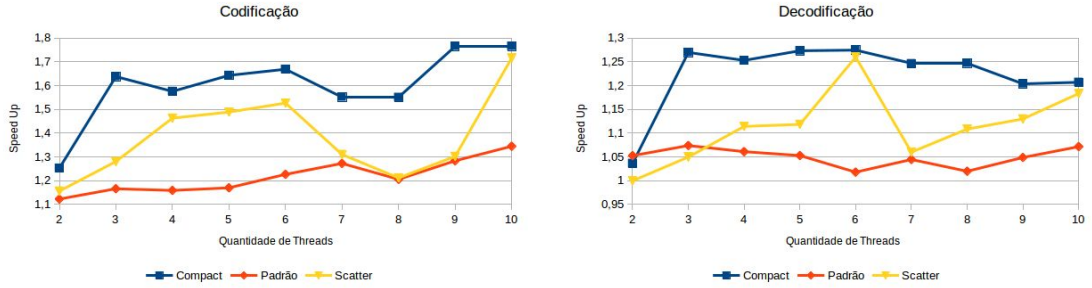

**Figura 3. Tempo de execução total do GMPR para textos**

## 4.2. Cripto-compressão de Imagens

A Figura 4a mostra a comparação entre as 3 estratégias de thread pinning para a codificação de imagens. Conforme é possível perceber,a estratégia que apresenta o melhor desempenho é a Compact, a qual fixa todas as threads na mesma CPU, distribuindo em ordem crescente entre os núcleos disponíveis.

A Figura 4b mostra a comparação entre as 3 estratégias de thread pinning para a decodificação de imagens. Conforme o gráfico mostra, a estratégia Compact novamente apresenta um desempenho melhor, sendo portanto a escolhida para a versão de imagensdo algoritmo GMPR.

A estratégia Compact apresentou os melhores resultados pelos mesmos motivos citados na versão de texto do algoritmo, reduzindo a latência no acesso à memória, devido à arquitetura NUMA da máquina experimental.

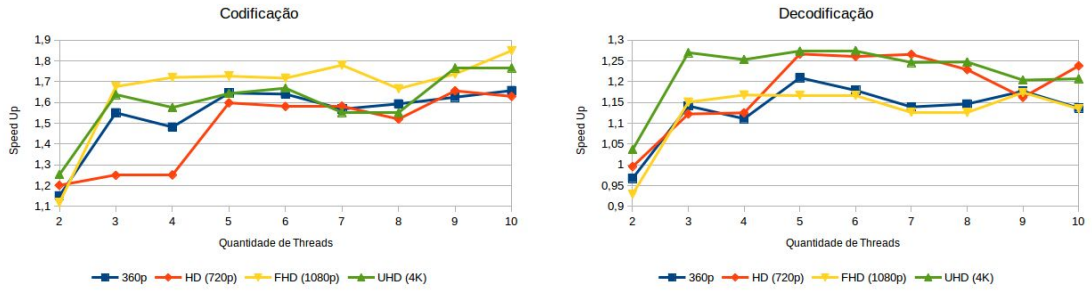



**(a) Codificação** **(b) Decodificação**

**Figura 4. Impacto das estratégias de thread pinning na cripto-compressão de imagens**

A Figura 5a mostra a comparação entre o tamanho das imagens e a quantidade de threads utilizadas para a codificação de imagens. Conforme exibido no gráfico, todos os tamanhos de imagem obtiveram um speedup crescente em relação ao número de threads utilizadas. O speedup tende a ser maior para imagens maiores.

A Figura 5b mostra a comparação entre o tamanho das imagens e a quantidade de threads utilizadas para a decodificação de imagens. Este gráfico mostra um desempenho bastante similar entre as diferentes imagens, sendo que todas obtiveram um speedup praticamente constante em relação à quantidade de threads.




**(a) Codificação** **(b) Decodificação**

**Figura 5. Impacto do tamanho dos arquivos na cripto-compressão de imagens**

Como se pode perceber no gráfico da Figura 6, a parte de codificação ganhou bastante desempenho na otimização sequencial e também no uso de paralelismo, enquanto a parte de decodificação apresentou um ganho mais tímido, obtendo uma performance melhor na versão paralela.

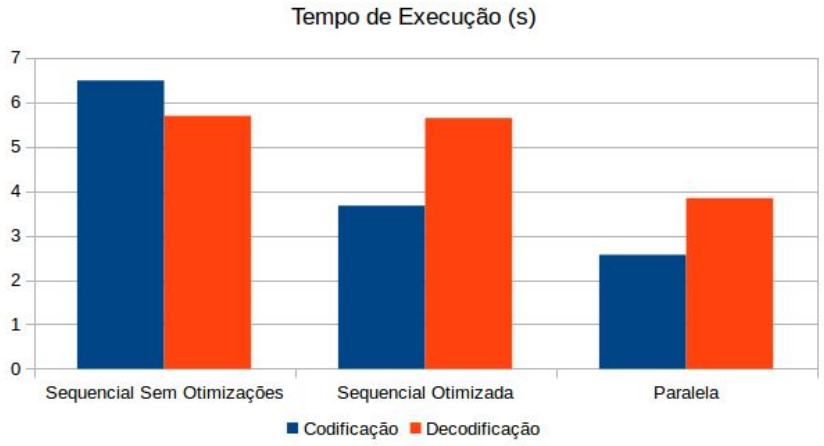



**Figura 6. Tempo de execução total do GMPR para imagens**

## 5. Conclusão e Trabalhos Futuros

O algoritmo GMPR funciona corretamente, cumprindo seu objetivo com maestria, porém apresenta um desempenho insatisfatório, principalmente na versão de textos. Este trabalho buscou analisar o código e melhorá-lo, garantindo que seu tempo de execução diminua e o torne possível de ser utilizado em aplicações reais.

A utilização da API OpenMP foi a solução escolhida para melhorar o desempenho do algoritmo de cripto-compressão GMPR, em conjunto com outras otimizações de código que não envolvem paralelismo. A implementação sequencial otimizada e a paralela tiveram seus resultados medidos e foram melhoradas gradativamente, de forma a garantir um ganho de performance.

Ter o algoritmo funcionando de forma eficiente é uma conquista muito interessante, pois o mesmo poderá ser utilizado nos aeroportos do Reino Unido sem se tornar um gargalo tanto em tempo de execução quanto no tamanho dos arquivos de imagem, reduzindo o armazenamento utilizado, os riscos de segurança e também o tempo necessário para enviar informações através da Internet.

Os resultados obtidos neste trabalho foram significantes, obtendo uma boa redução no tempo necessário para a execução do algoritmo GMPR. As execuções foram 1,47x mais rápidas na codificação e 11,9x na decodificação para a parte de textos, e 2,53x mais rápidas na codificação e 1,48x na decodificação para a parte de imagens.

Possíveis melhorias ou trabalhos futuros neste algoritmo poderiam envolver paralelismo em múltiplos computadores, através do MPI, ou a execução em GPUs, através do CUDA ou OPENCL. Também pode ser estudado um ganho de desempenho ao se usar uma implementação paralela da biblioteca usada no algoritmo de imagens, a OPENCV, assim como versões paralelas das funções nativas do C++, contidas na biblioteca padrão da linguagem.

O uso deste trabalho em outros projetos também é bastante possível, afinal esse algoritmo pode ser adaptado para o uso em projetos que tenham restrições de segurança das informações ou restrições de armazenamento. Há também a possibilidade desse projeto ser estendido para outras áreas, como a cripto-compressão de um banco de dados, dentre outras.

## Referências


RODRIGUES, M. A.; SIDDEQ, M. M. Information systems: Secure access and storage in the age of cloud computing. Athens Journal of Sciences, Athens Institute for Education and Research, v. 3, n. 4, p. 267–284, September 2016. Disponível em:<http://shura.shu.ac.uk/13715/>.

TANENBAUM, A. S.; BOS, H.Modern Operating Systems. Fourth. [S.l.]: Pearson, 2015.1137 p. ISBN 978-0-13-359162-0.

STALLINGS, W. Cryptography and Network Security: Principles and Practice. [S.l.]:Pearson, 2014. 731 p. ISBN 978-0133354690.

SALOMON, D. Data Compression: The Complete Reference. [S.l.]: Springer Science & Business Media, 2007. 1092 p. ISBN 978-1846286032.

CHANDRA, R. et al.Parallel Programming in OpenMP. [S.l.]: Morgan Kaufmann, 2000.231 p. ISBN 978-1558606715.

CHAPMAN, B.; JOST, G.; PAS, R.Using OpenMP: Portable Shared Memory Parallel Programming. [S.l.]: MIT Press, 2008. 353 p. ISBN 978-0262533027.