Unicamp - Instituto de Computação

MC920 / MO443

Introdução à Programação Paralela

Prof. Hervé Yviquel
1º semestre de 2023

Relatório do Projeto Final

Alunos:
George Gigilas Junior - 216741
Vinícius de Oliveira Peixoto Rodrigues - 245294

Descrição do Projeto

O algoritmo *gzip* é bastante utilizado para compressão de dados, sendo um algoritmo eficiente e sem perdas de informação. A partir dele, podemos representar o arquivo utilizando menos *bytes* (compressão), facilitando seu armazenamento e sua transmissão. Além disso, ele também permite a transformação dessa representação no arquivo original, o que caracteriza a descompressão. O gzip é construído a partir de uma combinação das codificações *LZ77* e de *Huffman*, algoritmos bastante conhecidos e amplamente utilizados.

Diante disso, pretendíamos paralelizar uma implementação serial do algoritmo *gzip*, buscando otimizar seu tempo de execução. Sabemos que isso é possível devido ao fato de que a codificação de *Huffman* é paralelizável[1]. Além disso, sabemos que já existe uma versão paralelizada do *gzip*, conhecida como *pgzip*[2], que pode nos trazer algumas técnicas a serem utilizadas e já demonstrou valores interessantes de *speedup* (para arquivos de mais de 1 Mb). No entanto, essa implementação é bastante complexa e seria inviável fazermos uma paralelização do *gzip* que tivesse um *speedup* minimamente interessante, no período de apenas um mês.

Por esse motivo, optamos por paralelizar um mecanismo que é utilizado internamente no *gzip*, assim como em outros métodos de compressão, que é a codificação de *Huffman*. Além de ser mais palpável dentro do tempo que temos nesta disciplina, ele acaba sendo mais útil pois é utilizado para diversas implementações e aplicações. Sendo assim, nós podemos paralelizamos a compressão e a descompressão deste algoritmo.

No entanto, a descompressão possui forte dependência e não pode ser facilmente paralelizada. O processo consiste em decodificar uma cadeia de *bits* codificada para obter os símbolos originais. Porém, para detectar qual parte da cadeia de *bits* determina o início de um novo símbolo, é preciso ter lido o símbolo anterior. Ou seja, este processo é naturalmente serial, com forte dependência do que já foi descomprimido.

Após várias pesquisas, concluímos que paralelizar a descompressão também não seria viável. Com isso, optamos por paralelizar a compressão, que possui menos dependência e é mais facilmente paralelizável. Para isso, fizemos uma implementação em C da codificação de *Huffman* (disponível no Github). Tendo em mente que já não seria possível paralelizar o *gzip* como um todo, nem paralelizar a descompressão do código de *Huffman*, achamos que deveríamos fazer nossa própria implementação, trazendo maior complexidade ao projeto.

Para paralelizar o código, utilizamos a API do OpenMP, conforme aprendemos em aula e utilizamos em alguns dos laboratórios. Com isso, esperamos obter bons valores de *speedup*, confirmando que a paralelização deste algoritmo pode trazer ganhos interessantes.

Nas seguintes seções deste relatório, vamos detalhar o funcionamento do algoritmo *gzip*, tanto a compressão quanto a descompressão, além de explicar noções básicas das codificações *LZ77* e de *Huffman*. Depois disso, vamos discutir e detalhar a implementação paralela que fizemos, passando por cada paralelização introduzida e suas peculiaridades. Feito isso, vamos passar pelos resultados que obtivemos ao fazer o perfilamento, analisando os trechos mais críticos do código, tanto para a versão serial, quanto para a nossa versão paralelizada. Por fim, faremos uma análise de desempenho da nossa versão, tendo em vista os resultados de *speedup* obtidos.

Detalhamento do Algoritmo

Como dito na seção anterior, o *gzip* é construído a partir da combinação das codificações *LZ77* e de *Huffman*. Vamos começar explicando como funciona a codificação de *Huffman*[3], para melhor entendimento do funcionamento do algoritmo *gzip*. A essência de um algoritmo de codificação de caracteres é poder representar um texto com menos *bytes*, de forma a diminuir seu tamanho, garantindo que ele pode ser decodificado para sua forma original. Diante disso, o código de *Huffman* calcula uma tabela de códigos que permite a maior compressão possível do texto original, dentre as compressões sem perda de informação.

Para entender como funciona essa codificação, precisamos conhecer o que é uma árvore de *Huffman*. A árvore de *Huffman* se trata de uma árvore binária em que cada nó interno (que não é folha) possui dois filhos. Define-se como frequência de um caractere, o número de vezes que ele aparece no texto a ser codificado. Cada nó da árvore possui um peso, representando a soma das frequências dos nós folha abaixo dele. Cada nó folha, por sua vez, corresponde a um caractere do texto original, juntamente com seu peso (frequência). Um exemplo de árvore de *Huffman* está presente na Figura 1, construída a partir do texto “this is an example of a huffman tree”. Podemos perceber que os nós nos níveis mais altos têm frequência maior, devido à forma como a árvore é estruturada. Esse detalhe é importante e ficará mais claro a seguir.

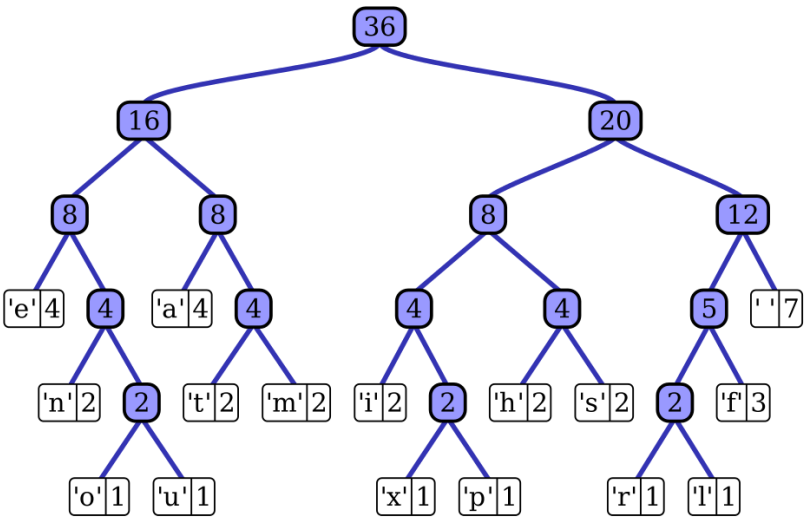

Figura 1: Árvore de Huffman construída a partir da sequência “this is an example of a huffman tree”.

A ideia do algoritmo de *Huffman* é codificar os símbolos do alfabeto como o caminho a ser percorrido da raiz da árvore de *Huffman* até o símbolo (usando espaço logarítmico em relação ao número de símbolos). Esse procedimento é feito a partir da construção de uma tabela de códigos, em que um código é atribuído a cada símbolo. É importante ressaltar que essa tabela é livre de prefixos, ou seja, em uma sequência desses códigos, a identificação dos códigos não é ambígua, pois um código não é prefixo de outro. Os códigos 0, 1 e 01 não são livres de prefixo, por exemplo, pois, em uma sequência 01, não conseguimos distinguir se a sequência é composta por 0 e 1, ou por 01.

Quanto ao fato de os níveis mais altos possuírem uma frequência maior, isso é uma forma de garantir que os símbolos mais frequentes possuirão uma codificação menor (em número de *bits*). Ao atribuir os códigos para cada símbolo, esse critério é levado em

consideração, pois o código final ficará menor desta forma. Desta forma, o código de *Huffman* pode garantir a maior compressão possível para um texto, a partir da tradução dos símbolos utilizando a tabela de códigos.

Para a mesma frase que produziu a árvore de Huffman da Figura 1, criamos a tabela de código (Tabela 1). A coluna “Economia” foi calculada assumindo que cada caractere é representado por 8 bits. A partir dela, obtemos a seguinte codificação binária: 100010100011101111100111011111011010011100000100011100100101110010000111010101101111011111101001011110111011001011010011111000110000000000.

Tabela 1: Tabela de códigos construída a partir da sequência “this is an example of a huffman tree”.

Caractere	Frequência	Código	Economia
espaço	7	111	7*8 - 7*3 = 35 bits
a	4	011	4*8 - 4*3 = 20 bits
e	4	000	4*8 - 4*3 = 20 bits
f	3	1101	3*8 - 3*4 = 12 bits
t	2	1000	2*8 - 2*4 = 8 bits
h	2	1010	2*8 - 2*4 = 8 bits
i	2	0011	2*8 - 2*4 = 8 bits
s	2	1011	2*8 - 2*4 = 8 bits
n	2	0100	2*8 - 2*4 = 8 bits
m	2	1001	2*8 - 2*4 = 8 bits
x	1	00100	1*8 - 1*5 = 3 bits
p	1	00101	1*8 - 1*5 = 3 bits
l	1	11001	1*8 - 1*5 = 3 bits
o	1	01010	1*8 - 1*5 = 3 bits
u	1	01011	1*8 - 1*5 = 3 bits
r	1	11000	1*8 - 1*5 = 3 bits

Para decodificar uma cadeia de bits codificados dessa forma, o processo é bastante simples. Basta ter a árvore de *Huffman* e percorrê-la da raiz até as folhas, de acordo com o próximo bit lido: o bit 0 leva a um dos filhos do nó (geralmente o esquerdo), enquanto o bit 1 leva ao outro filho. Ao encontrar um nó-folha, um símbolo foi decodificado. Então, basta voltar à raiz da árvore e repetir o processo até acabar a cadeia de bits. Ao final desse processo, você terá a mensagem decodificada.

Agora, vamos apresentar a codificação *LZ77*[4], criada em 1977 por Abraham Lempel e Jacob Ziv, que também é um algoritmo de compressão sem perdas de informação. Tal algoritmo se baseia na utilização de dicionários para armazenar sequências de símbolos

lidas do texto original. Assim, ao encontrar uma nova ocorrência desta mesma sequência, ela é substituída pela posição de sua última ocorrência. Com o intuito de limitar o espaço de busca por ocorrências e o endereçamento, existe um intervalo máximo para o qual as ocorrências anteriores são mantidas.

Indo mais a fundo na codificação do *LZ77*, inicialmente o *buffer* (que contém parte do texto a ser codificado) e o dicionário estão vazios. Então, parte dos dados entram no *buffer* e, à medida em que são codificados, eles vão sendo adicionados ao dicionário. Após isso, novos dados são adicionados ao *buffer* e tudo se repete, até acabar o texto. Quando uma sequência encontrada já está presente no dicionário, ela é substituída por um *token* do tipo (P, T, C), em que P corresponde à posição, no *buffer*, do primeiro símbolo da sequência encontrada, T corresponde ao comprimento desta sequência e C é o primeiro símbolo do *buffer* que não está presente na sequência.

Para fazer a decodificação, basta percorrer o texto codificado e ir construindo o dicionário. Ao ler um *token*, se existir a coincidência com alguma sequência anterior, é feita a reconstrução da sequência através dos dados P, T e C. Caso contrário, significa que o *token* não corresponde à uma sequência já encontrada e, portanto, basta substituí-lo pelo símbolo que está na posição P do *buffer* e adicionar a sequência no dicionário (para que, no futuro, ela possa ser utilizada). Na Figura 3, temos um exemplo ilustrativo da codificação *LZ77* aplicada ao texto da Figura 2.

001:001 In the beginning God created the heaven and the earth.

001:002 And the earth was without form, and void; and darkness was upon the face of the deep. And the Spirit of God moved upon the face of the waters.

Figura 2: Exemplo de texto correspondente a um trecho retirado da Bíblia.

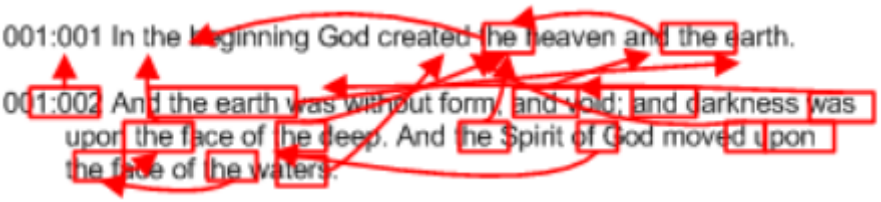

Figura 3: Exemplo ilustrativo da codificação *LZ77* aplicada no texto da Figura 2.

Agora, enfim, vamos partir para a descrição do funcionamento do *gzip*[5]. Este algoritmo é uma codificação de tamanho variável, assim como a codificação de *Huffman* (cada código pode ter um tamanho diferente, em *bytes*). De maneira geral, a primeira etapa do algoritmo consiste em comprimir a mensagem de entrada com a codificação *LZ77*. Por sua vez, a saída dessa codificação é codificada com a codificação de *Huffman*, possibilitando que os ponteiros (de *LZ77*) e os demais símbolos possam ser misturados e ainda serem decodificados corretamente.

Considerando que mensagens diferentes geram tabelas de código de *Huffman* diferentes, o algoritmo *gzip* requer que o código comprimido produza uma tabela única para cada mensagem comprimida. Então, uma outra etapa requer que essa tabela seja comprimida (e adicionada na mensagem final) utilizando, também, a codificação de *Huffman*. Vale ressaltar que o algoritmo implementa condições adicionais aos códigos produzidos, de forma que os códigos são padronizados. Assim, não é necessário construir a

tabela da maneira tradicional, basta seguir os códigos padronizados e especificar, para cada símbolo, o tamanho do seu código correspondente. Com isso, é possível transmitir a tabela de código utilizando menos *bits*, o que é desejado na compressão de uma mensagem.

No entanto, apesar dessa técnica de compressão, a tabela gerada ainda é bastante grande, pois deve conter um código para cada símbolo e cada ponteiro. Por isso, o tamanho do código de cada símbolo (que agora substitui o código do símbolo em si, como visto no parágrafo anterior) também é codificado a partir da codificação de *Huffman*. Para evitar mais passos, os tamanhos desses novos códigos possuem valores fixos de 3 *bits*. Para exemplificar, uma sumarização de tudo isso que foi falado está ilustrado na Figura 4. Adicionalmente, existe um campo no cabeçalho dos dados que indica quantos códigos de *Huffman* estão presentes em cada tabela de códigos, para possibilitar a interpretação correta delas.

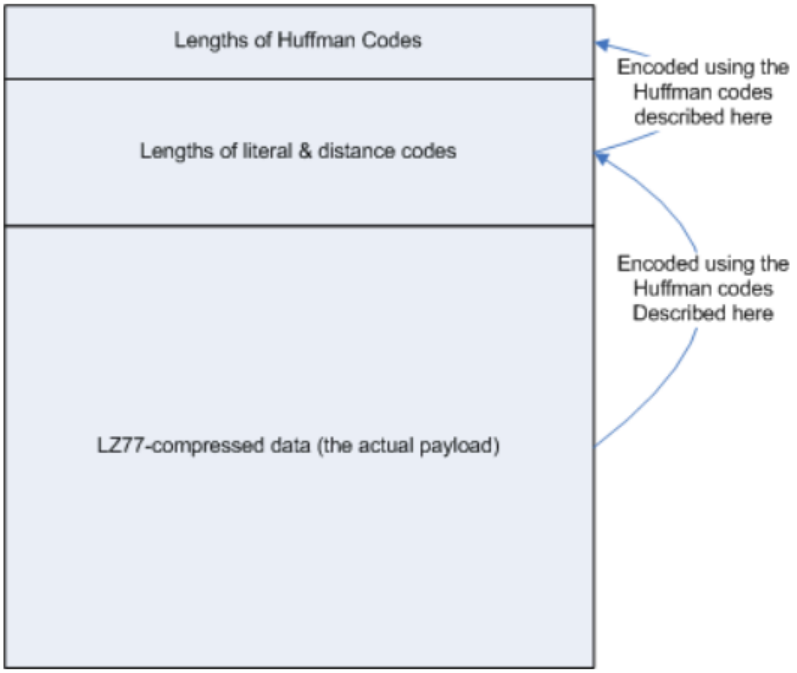

Figura 4: Exemplificação do processo de codificação, com proporções ilustrativas do tamanho de cada conjunto de dados.

Um aspecto importante a ser destacado é o empacotamento dos *bits* dos códigos em *bytes*, pelo fato de a codificação de tamanho variável não garantir que cada *byte* corresponda a um símbolo. Os conteúdos (expressos pelos símbolos) são interpretados com o formato *big-endian* (*bit* mais significativo à esquerda). No entanto, quando mais de um conteúdo é empacotado em um mesmo *byte*, isso é feito no formato *little-endian* (*bit* mais significativo à direita). Por exemplo, os códigos 001 (3 *bits*) e 11100 (5 *bits*), se forem lidos nesta ordem (primeiro 001 e depois 11100), serão empacotados como 11100001 (ordem inversa, porém, dentro de cada código, a ordem não mudou). Isso traz uma lógica adicional a ser tratada na implementação.

Para decodificar a mensagem, em linhas gerais, o processo consiste em três etapas e é bastante intuitivo. Primeiramente, é necessário ler os códigos de *Huffman* que descrevem a tabela de códigos que codificou o conteúdo da mensagem. Após isso, basta utilizar esses códigos para decodificar a tabela de códigos que codificou o conteúdo da mensagem. Por fim, deve-se utilizar essa tabela de códigos final para decodificar o conteúdo da mensagem comprimida, lendo um símbolo por vez. Então, se o símbolo for um

literal, basta imprimi-lo; se o símbolo for um ponteiro, basta imprimir o conjunto de caracteres ao qual ele se refere (como o ponteiro aponta sempre para trás, esse conjunto de caracteres já foi impresso).

O formato de arquivo GZIP possui um cabeçalho e uma série de blocos (que contêm blocos de mensagens comprimidas). Em cada bloco, existe um campo que especifica o seu formato: o primeiro *bit* indica se o bloco é o último (se sim, é 1; se não, é 0) e os dois *bits* seguintes indicam o tipo do bloco (00 = não comprimido, 01 = comprimido com código de *Huffman* fixo, 10 = comprimido com código de *Huffman* dinâmico, 11 é um tipo inválido). Com isso, o bloco pode ser descomprimido de acordo.

Finalmente, vale notar que o GZIP especifica que os documentos possuam, no final, um *Cyclic Redundancy Check* (CRC) e o tamanho da mensagem não codificada. O CRC é uma espécie de *checksum*, mas mais robusto e confiável, menos suscetível a indicar falso positivo (por exemplo quando um segundo *bit* errado faz com que o *checksum* coincida com o valor correto esperado). O objetivo é detectar se houve algum erro que possa ter invalidado a leitura da mensagem original.

Descrição da Implementação

A implementação da codificação de *Huffman* segue os princípios que foram descritos na seção anterior. Neste projeto, foram implementadas a versão serial e a versão paralela usando *OpenMP*. Para facilitar a execução dos códigos (tanto em paralelo quanto serial), preparamos um arquivo *cmake*, similar ao que foi feito nos laboratórios. As instruções para executar os programas estão no README do projeto.

Quanto à versão serial, criamos três arquivos: huffman.c, minheap.c e serial_compression.c. O primeiro deles implementa as principais funcionalidades relacionadas à árvore de *Huffman* e à tabela de códigos, sendo elas: a estrutura de dados da árvore (que contém um ponteiro para a raiz da árvore, o tamanho da árvore e um *heap* de mínimo), a estrutura de dados do código em si (que contém o código associado a um símbolo e o seu tamanho), uma função para gerar uma árvore (que recebe um vetor de frequências de símbolos), uma função para destruir a árvore, uma função para gerar a tabela de códigos (que recebe um ponteiro para a árvore e um dicionário da *struct* de código), e uma função para imprimir a árvore.

O segundo arquivo implementa o *heap* de mínimo utilizado para manipular os nós da árvore. Aqui, definimos um *enumerate* para indicar o tipo de nó, que pode ser vazio, um nó interno ou um nó folha. Ademais, definimos duas estruturas de dados: o nó da árvore (que contém seu tipo, a frequência associada a ele, o símbolo associado a ele - se for folha -, e um apontador para cada um de seus dois filhos). Por fim, definimos algumas funções: criação de destruição de nós, criação de um novo *heap* de mínimo, uma função que remove o próximo nó do *heap*, inserção de nó no *heap*, destruição do *heap* e impressão dele.

O terceiro e último arquivo é responsável por utilizar os mecanismos auxiliares dos demais arquivos para realização da compressão em si. Para isso, definimos duas estruturas de dados: *bitstream* (que contém um *buffer*, um valor de capacidade, um valor de tamanho e um valor de *offset*), para podermos manipular os dados em nível de *bit*, como especificado para algoritmos de compressão; e o *serial_compressor* (que contém um dicionário de códigos - que é a tabela de códigos -, um ponteiro para o arquivo de entrada, o tamanho do arquivo de entrada e um *bitstream* - que conterá o código comprimido -), responsável por fazer, de fato, a compressão.

O fluxo da compressão começa com a construção da tabela de frequências, contando a quantidade de vezes que cada símbolo aparece no arquivo de entrada. Feito isso, é feita a construção da árvore de *Huffman* em si e a tabela de códigos. Com essas estruturas prontas, o arquivo de entrada, então, é processado e cada símbolo é lido. A cada leitura, o código correspondente ao símbolo (lido da tabela de códigos) é adicionado ao *bitstream*. Após todas as iterações, o código estará completo na *bitstream*.

A construção da árvore de *Huffman* não é facilmente paralelizável, pois os nós precisam ser construídos e conectados um por vez. No entanto, este processo é feito uma vez por compressão, e não deve ser um gargalo. As outras etapas, entretanto, podem ser paralelizadas como será descrito nos parágrafos a seguir.

A primeira parte paralelizada do código foi a geração da tabela de frequências, que posteriormente é necessária para construir a árvore de *Huffman*. Ela é construída como se fosse um histograma: cada símbolo da mensagem original é percorrido e incrementa em 1 a sua contagem. Para paralelizar isso, utilizamos a diretiva do *parallel* e a diretiva do *for* e, ao incrementar em 1 a contagem de cada símbolo, percebemos que haveria uma condição de corrida (mais de uma *thread* poderia tentar ler e escrever ao mesmo tempo).

A diretiva do *parallel* inicializa o número máximo de *threads* possível (através do método `omp_get_max_threads()`) e define as variáveis privadas e compartilhadas. Dentre as compartilhadas, temos o vetor de frequências e o ponteiro para a *struct parallel_compressor*, que são compartilhadas pois todas as *threads* necessitam acessar e atualizar as frequências e utilizar o mesmo compressor.

Após isso, temos a diretiva *for*, antes do *loop* que faz a contagem das frequências, com *schedule static*, pois trouxe bons desempenhos em nossos testes. Essa diretiva é responsável por distribuir as iterações do *loop* para as *threads*, com número aproximadamente igual de iterações por *thread*. Dentro do *loop*, atualizamos um vetor de frequência local, a fim de evitar a condição de corrida ao tentar incrementar em 1 a frequência. Então, após o *loop* (e ainda dentro da diretiva *parallel*), fazemos a junção dos vetores locais de frequência e adicionamos a diretiva *critical* para garantir que apenas uma *thread* faria isso. Assim, garantimos que a construção da tabela de frequência está correta e paralelizada

A segunda parte a ser paralelizada é a geração do código de *Huffman* em si, que adiciona o resultado à *bitstream*. Aqui, seria muito difícil coordenar em qual posição de cada *byte* cada *thread* deveria escrever seu resultado. Para isso, fizemos cada *thread* escrever seu resultado em um *buffer* diferente e, no final de tudo, juntamos esses *buffers* em uma mensagem só, tomando cuidado para alinhar os resultados dentro dos *bytes* calculando o *offset* correto.

Para fazer o *buffer* local, utilizamos a diretiva *parallel*, especificando o número máximo de *threads* disponível na máquina testada (com o mesmo método citado anteriormente). Então, adicionamos uma diretiva *for* com *schedule static* antes do *loop* que percorre os símbolos do arquivo de entrada para transformá-los nos códigos de *Huffman* respectivos. Com isso, garantimos uma paralelização dessa etapa, mantendo o código correto e sem condições de corrida, uma vez que os cálculos são feitos de maneira isolada e depois fundidos por uma única *thread*.

Perfilamento

Nesta seção, vamos analisar o perfilamento para ambas as versões que produzimos, obtido a partir da ferramenta *perf*. Na Figura 5, temos o resultado para a implementação serial e, na Figura 6, o resultado para a implementação paralela, ambos para os melhores resultados de *speedup* que obtivemos (maior arquivo de entrada).

```
Performance counter stats for 'build/serial_compression /home/nuke/IINACT/Network_26701_20230104.log' (5 runs):

         3,136.10 msec task-clock:u                     #    4.423 CPUs utilized             ( +- 14.15% )
                0      context-switches:u               #    0.000 /sec
                0      cpu-migrations:u                 #    0.000 /sec
            3,572      page-faults:u                    #    1.922 K/sec                     ( +- 15.45% )
   12,441,960,273      cycles:u                         #    6.695 GHz                       ( +- 14.17% )  (83.15%)
      765,128      stalled-cycles-frontend:u          #    0.01% frontend cycles idle      ( +- 13.57% )  (83.30%)
    9,559,209,368      stalled-cycles-backend:u         #  129.73% backend cycles idle       ( +- 14.15% )  (83.46%)
   28,700,237,033      instructions:u                   #    3.89  insn per cycle
                                                  #    0.20  stalled cycles per insn     ( +- 14.12% )  (83.31%)
    1,041,051,899      branches:u                       #  560.218 M/sec                     ( +- 14.14% )  (83.65%)
           16,721      branch-misses:u                  #    0.00% of all branches           ( +- 14.29% )  (83.13%)

          0.7090 +- 0.0320 seconds time elapsed  ( +-  4.52% )
```

Figura 5: Resultado do perfilamento do teste 4 para nossa implementação serial da compressão de *Huffman*.

```
Performance counter stats for 'build/parallel_compression /home/nuke/IINACT/Network_26701_20230104.log' (5 runs):

        11,596.19 msec task-clock:u                     #   30.378 CPUs utilized             ( +- 13.90% )
                0      context-switches:u               #    0.000 /sec
                0      cpu-migrations:u                 #    0.000 /sec
           22,587      page-faults:u                    #    3.204 K/sec                     ( +- 14.22% )
   46,678,178,154      cycles:u                         #    6.622 GHz                       ( +- 13.90% )  (83.06%)
    3,712,810,346      stalled-cycles-frontend:u        #   13.11% frontend cycles idle      ( +- 14.54% )  (83.52%)
   10,528,309,570      stalled-cycles-backend:u         #   37.17% backend cycles idle       ( +- 14.42% )  (83.96%)
   36,420,804,703      instructions:u                   #    1.29  insn per cycle
                                                  #    0.17  stalled cycles per insn     ( +- 14.16% )  (83.56%)
    2,226,948,141      branches:u                       #  315.943 M/sec                     ( +- 14.12% )  (82.98%)
           81,525      branch-misses:u                  #    0.01% of all branches           ( +- 13.73% )  (82.98%)

          0.3817 +- 0.0107 seconds time elapsed  ( +-  2.81% )
```

Figura 6: Resultado do perfilamento do teste 4 para nossa implementação paralela da compressão de *Huffman*.

Comparando as Figuras 5 e 6, notamos que a eficiência da implementação paralela já se tornou significativamente maior que a serial, evidenciando que é mais vantajoso usar a implementação paralela para comprimir um volume de dados maior.

Abaixo, seguem as demais Figuras: as pares correspondem à implementação serial, enquanto as ímpares correspondem à implementação paralela. A partir delas, podemos perceber que a tendência é parecida, com a diferença que, para arquivos menores (Figuras 7 a 10), o *overhead* associado ao paralelismo se faz mais presente, deixando o tempo de execução maior no caso paralelo. Com isso, concluímos que a implementação paralela é benéfica, mas apenas para arquivos maiores (acima de 1 Mb, ou perto disso).

```
Performance counter stats for 'build/serial_compression data/lorem.txt' (5 runs):

              1.91 msec task-clock:u                     #    2.756 CPUs utilized             ( +- 12.76% )
                 0      context-switches:u               #    0.000 /sec
                 0      cpu-migrations:u                 #    0.000 /sec
               366      page-faults:u                    #  299.537 K/sec                     ( +- 14.30% )
         2,020,041      cycles:u                         #    1.653 GHz                       ( +- 14.25% )  (81.73%)
           125,476      stalled-cycles-frontend:u        #   10.39% frontend cycles idle      ( +- 13.05% )
           809,146      stalled-cycles-backend:u         #   67.01% backend cycles idle       ( +- 12.12% )
         2,777,516      instructions:u                   #    2.30  insn per cycle
                                                  #    0.19  stalled cycles per insn     ( +- 14.14% )
           377,296      branches:u                       #  308.782 M/sec                     ( +- 14.14% )
             5,387      branch-misses:u                  #    2.38% of all branches           (18.27%)

         0.0006943 +- 0.0000533 seconds time elapsed  ( +-  7.67% )
```

Figura 7: Resultado do perfilamento do teste 1 para nossa implementação serial da compressão de *Huffman*.

```
Performance counter stats for 'build/parallel_compression data/lorem.txt' (5 runs):

          300.22 msec task-clock:u                     #   59.226 CPUs utilized             ( +-  4.05% )
               0      context-switches:u               #    0.000 /sec
               0      cpu-migrations:u                 #    0.000 /sec
             610      page-faults:u                    #    2.289 K/sec                      ( +- 14.18% )
   1,175,192,083      cycles:u                         #    4.410 GHz                        ( +-  3.22% )  (82.67%)
     349,386,533      stalled-cycles-frontend:u        #   33.00% frontend cycles idle       ( +-  3.21% )  (86.10%)
      78,255,728      stalled-cycles-backend:u         #    7.39% backend cycles idle        ( +-  6.62% )  (86.15%)
     130,962,465      instructions:u                   #    0.12  insn per cycle
                                                  #    2.33  stalled cycles per insn    ( +-  4.08% )  (87.29%)
      36,766,448      branches:u                       #  137.970 M/sec                      ( +-  3.95% )  (94.45%)
          15,963      branch-misses:u                  #    0.05% of all branches            ( +- 16.82% )  (75.64%)

        0.00507 +- 0.00302 seconds time elapsed  ( +- 59.52% )
```

Figura 8: Resultado do perfilamento do teste 1 para nossa implementação paralela da compressão de *Huffman*.

```
Performance counter stats for 'build/serial_compression data/macbeth.txt' (5 runs):

            9.62 msec task-clock:u                     #    4.271 CPUs utilized             ( +- 14.04% )
               0      context-switches:u               #    0.000 /sec
               0      cpu-migrations:u                 #    0.000 /sec
             462      page-faults:u                    #   78.702 K/sec                     ( +- 14.08% )
      30,968,537      cycles:u                         #    5.276 GHz                       ( +- 13.82% )  (96.11%)
         139,687      stalled-cycles-frontend:u        #    0.74% frontend cycles idle      ( +- 13.31% )
      22,193,262      stalled-cycles-backend:u         #  117.64% backend cycles idle       ( +- 14.13% )
      67,524,472      instructions:u                   #    3.58  insn per cycle
                                                  #    0.20  stalled cycles per insn    ( +- 14.14% )
       2,761,824      branches:u                       #  470.480 M/sec                     ( +- 14.14% )
           1,881      branch-misses:u                  #    0.11% of all branches           (3.89%)

       0.0022521 +- 0.0000453 seconds time elapsed  ( +-  2.01% )
```

Figura 9: Resultado do perfilamento do teste 2 para nossa implementação serial da compressão de *Huffman*.

```
Performance counter stats for 'build/parallel_compression data/macbeth.txt' (5 runs):

          181.43 msec task-clock:u                     #   55.542 CPUs utilized             ( +- 15.79% )
               0      context-switches:u               #    0.000 /sec
               0      cpu-migrations:u                 #    0.000 /sec
             998      page-faults:u                    #    7.569 K/sec                     ( +- 14.13% )
     604,329,609      cycles:u                         #    4.584 GHz                       ( +- 17.74% )  (64.51%)
     119,809,435      stalled-cycles-frontend:u        #   27.65% frontend cycles idle      ( +- 18.42% )  (75.26%)
      76,231,538      stalled-cycles-backend:u         #   17.59% backend cycles idle       ( +- 14.82% )  (93.64%)
     154,933,745      instructions:u                   #    0.36  insn per cycle
                                                  #    0.61  stalled cycles per insn    ( +- 14.52% )
      25,313,081      branches:u                       #  191.986 M/sec                     ( +- 15.52% )
          12,976      branch-misses:u                  #    0.07% of all branches           ( +- 12.27% )  (90.31%)

        0.00327 +- 0.00138 seconds time elapsed  ( +- 42.22% )
```

Figura 10: Resultado do perfilamento do teste 2 para nossa implementação paralela da compressão de *Huffman*.

```
Performance counter stats for 'build/serial_compression /home/nuke/livro_chines.pdf' (5 runs):

          100.14 msec task-clock:u                     #    4.779 CPUs utilized             ( +- 14.00% )
               0      context-switches:u               #    0.000 /sec
               0      cpu-migrations:u                 #    0.000 /sec
           2,265      page-faults:u                    #   37.471 K/sec                     ( +- 14.13% )
     368,923,878      cycles:u                         #    6.103 GHz                       ( +- 13.90% )  (82.76%)
         244,680      stalled-cycles-frontend:u        #    0.11% frontend cycles idle      ( +- 13.02% )  (85.03%)
     269,652,100      stalled-cycles-backend:u         #  120.06% backend cycles idle       ( +- 14.39% )  (85.03%)
     822,965,939      instructions:u                   #    3.66  insn per cycle
                                                  #    0.20  stalled cycles per insn    ( +- 14.17% )  (85.05%)
      31,324,847      branches:u                       #  518.222 M/sec                     ( +- 14.12% )  (85.03%)
          20,000      branch-misses:u                  #    0.11% of all branches           ( +- 14.08% )  (77.10%)

        0.020954 +- 0.000906 seconds time elapsed  ( +-  4.33% )
```

Figura 11: Resultado do perfilamento do teste 3 para nossa implementação serial da compressão de *Huffman*.

```
Performance counter stats for 'build/parallel_compression /home/nuke/livro_chines.pdf' (5 runs):

        1,221.20 msec task-clock:u                     #   61.565 CPUs utilized             ( +- 14.96% )
               0      context-switches:u               #    0.000 /sec
               0      cpu-migrations:u                 #    0.000 /sec
           4,354      page-faults:u                    #    5.783 K/sec                     ( +- 14.14% )
   4,994,046,808      cycles:u                         #    6.634 GHz                       ( +- 14.90% )  (77.54%)
     999,736,508      stalled-cycles-frontend:u        #   32.37% frontend cycles idle      ( +- 18.00% )  (77.47%)
     429,617,118      stalled-cycles-backend:u         #   13.91% backend cycles idle       ( +- 13.77% )  (81.41%)
   1,558,205,111      instructions:u                   #    0.50  insn per cycle
                                                  #    0.40  stalled cycles per insn    ( +- 14.08% )  (86.24%)
     173,061,957      branches:u                       #  229.877 M/sec                     ( +- 14.90% )  (90.32%)
          27,681      branch-misses:u                  #    0.03% of all branches           ( +- 14.74% )  (87.82%)

        0.01984 +- 0.00330 seconds time elapsed  ( +- 16.66% )
```

Figura 12: Resultado do perfilamento do teste 3 para nossa implementação paralela da compressão de *Huffman*.

Análise de Desempenho

Para analisar o desempenho, executamos o código 5 vezes, calculando a média e o desvio padrão, considerando ambas implementações. Os testes foram executados em um Ryzen 7 3700X @ 4.2 GHz (8 cores/16 threads) com 32GB@3200MHz de RAM.

Seguem os testes executados abaixo, com entradas de tamanhos diferentes.

Primeiro teste: arquivo de 2 kb

Resultado da implementação serial:

- Média: 0,0006943 s
- Desvio padrão: 0,000053 s

Resultado da implementação paralela:

- Média: 0,00507 s
- Desvio padrão: 0,00302 s

Speedup: 0,136943

Segundo teste: arquivo de 117 kb

Resultado da implementação serial:

- Média: 0,0022521 s
- Desvio padrão: 0,0000453s

Resultado da implementação paralela:

- Média: 0,00327 s
- Desvio padrão: 0,00138 s

Speedup: 0,688715

Terceiro teste: arquivo de 1,5 Mb

Resultado da implementação serial:

- Média: 0,020954 s
- Desvio padrão: 0,000906 s

Resultado da implementação paralela:

- Média: 0,01984 s
- Desvio padrão: 0,00330 s

Speedup: 1,056149

Quarto teste: arquivo de 50 Mb

Resultado da implementação serial:

- Média: 0,7090 s
- Desvio padrão: 0,0320 s

Resultado da implementação paralela:

- Média: 0,3817 s
- Desvio padrão: 0,0107 s

Speedup: 1,8574797

Com base nos resultados obtidos, concluímos que eles foram bastante satisfatórios. Em especial, notamos que os resultados apresentaram bom *speedup* para os arquivos maiores (pelo menos acima de 1 Mb), pois apresentaram um *speedup* considerável entre 1,056149 e 1,8574797. Esses valores são bastante razoáveis e indicam que o problema de compressão a partir da codificação de *Huffman* pode ter benefícios de desempenho com paralelização. Em especial, notamos que, quanto maior o arquivo, maior o *speedup*, enquanto para os arquivos menores, os resultados não são tão bons. Isso ocorre pois o *overhead* associado à paralelização não compensa o tempo que demora para a execução serial para textos menores. Já para textos maiores, esse tempo gasto para paralelizar acaba compensando o tempo de execução como um todo. De maneira geral, esses resultados foram bastante conclusivos e indicam que essa técnica pode ser levada para os demais algoritmos que utilizam esse tipo de codificação.

Referências

[1] algorithm - Is there any way to parallelize Huffman encoding implementation on hardware? - Stack Overflow

[2] GitHub - klauspost/pgzip: Go parallel gzip (de)compression

[3] Código de Huffman para compressão de arquivos de texto (usp.br)

[4] LZ77 | Multimedia (ufp.pt)

[5] infinitepartitions.com/art001.html