Password Cracking Processamento Paralelo

André Thiago Borghi Couto, *Graduando, UFES*
Bruno Calmon Barreto, *Graduando, UFES*

Resumo—Passwords are currently one of the most important things for every human being living in this world, knowing that with one of these passwords it is possible to have all the information of a person's life in their hands, malicious people may want to take advantage of this with all methods encryption algorithms that currently exist that try to make us safer for attacks, but how to do this, how to find an encrypted password? In this work we will present a way to obtain certain passwords.

Index Terms—Computer Science, Parallel Processing, Comunnications, Password Cracking, LaTeX, Ciência da Computação, MPI, Processamento Paralelo.

1 Introdução

Se pudéssemos descrever um problema desafiador em nossa vida, esse problema seria lembrar todas as senhas, que atualmente são de suma importância para nosso cotidiano, seja virtualmente (redes sociais, jogos, aplicativos) ou reais (bancos ou cadeados), mesmo tomando todos os cuidados ainda estamos vulneráveis aos ataques de quebra (cracking) de senhas, assim, todo cuidado é pouco, pois se temos senhas simples, ou muito usadas, elas podem ser facilmente descobertas.

O trabalho em si é um exemplo de ataque de força bruta, no qual é aplicado sequencialmente todas as permutações de uma determinada quantidade de caracteres que por sua vez é criptografada e comparada à que foi passada para ser descoberta.

2 Estratégia

Primeiramente nosso código foi escrito em C++, utilizando a biblioteca MPI e técnicas da mesma, que são para a paralelização do código em nível de processo, como o fork.

Nossa estratégia consiste em dividir em um limite dentro do vetor dos caracteres, utilizando a divisão pela quantidade de Processadores para definir o início de cada um deles, assim cada processador recebe ▪ no qual devemos discutir abaixo pelo código. Mais adiante teremos as opções para discutir, uma delas leva em consideração que o $Processador\ 0$, não faz os cálculos, então deve-se admitir que cada Processador recebe ▪.

Para começarmos temos o básico em receber os dados de entrada ou seja a senha que vai ser quebrada, em seguida criamos a String de resposta, juntamente com a declaração dos caracteres que serão utilizados, isso permite definir qual a complexidade da senha a ser quebrada, logo para este trabalho utilizaremos o básico, alfabeto minúsculo e maiúsculo além dos números. Em seguida temos opções do MPI, que são np, como número de processadores e $rank$ como o número do atual processador, sendo o principal 0 e os demais distribuídos de forma incremental. Visto que MPI possui seus próprios métodos para manipulação de processos, a MPI_Init representa o início do código para o MPI, MPI_Comm_rank representa o numero do processador atual, e MPI_Comm_size representa a quantidade de processadores, no caso essa informação é externa, que é passada na execução do $mpirun - np\ \#$, onde $\#$ é o número desejado de processadores.

2.1 Main

Seguindo o código anterior, temos a verificação do processador que está sendo executado, neste caso o $Processador\ 0$, que é o primeiro, que suporta a base dos processos, e centraliza as operações como um sistema de $mestre/escravo$ que distribui tarefas do $mestre$ para os todos os demais $escravos$.

Começando temos a alocação do vetor $limites$ que vai armazenar os limites de início e fim que cada processador irá utilizar nas operações, em seguida a chamada da função $distribui$ que faz a divisão retornando para os $limites$, tendo esses valores, iniciamos a distribuição para os demais processadores, com um for, com o MPI_Send, enviamos os dados necessários ($limites$) para os demais processadores.

Cada processador segue com seus cálculos, até o ponto que encontra uma solução, caso isso aconteça, temos a continuação do código que está bloqueado, devido ao MPI_Probe, que está ai pra resolver um dos problemas de sincronismo, neste caso, é do tamanho do valor de retorno, pois podem ser retornadas Strings de tamanho $1\ a\ 7$, então não podemos criar um MPI_Recv para cada, por que não sabemos qual o tamanho da String, então como qualquer tamanho está sendo recebido, a função recolhe o status,

- *A.T.B. Couto está graduando em ciência da computação, na UFES, campus CEUNES, em São Mateus - ES.*
  E-mail: andreww.max@hotmail.com
- *B.C. Barreto está graduando em ciência da computação, na UFES, campus CEUNES, em São Mateus - ES.*
  *E-mail: bruno**********

Trabalho escrito em 1 de julho de 2019;

e o MPI_Get_count faz a leitura do tamanho da String sendo recebida, logo em seguida alocamos o tamanho à variável $RESPOSTA$, invocamos o MPI_Recv, que irá armazenar a resposta, e com o recebimento concluído, temos a exibição do Processador que encontrou junto com a senha descriptografada.

Uma diferença dos demais métodos é o desvio para o termino, não sendo muito elegante, ou faltando opções, optamos pelo MPI_Abort que finaliza um determinado grupo de comunicação, logo ao invocar a função temos o termino da execução de todas os processadores, atua como um $MPI\_Finalize$ forçado, por isso a execução pode apresentar a mensagem de erro ao terminar o programa, então caso for executar, para ocultar a informação de erro na saída, utilize a flag no formato $[-np\ \#\ -quiet]$.

Importante ressaltar que um dos problemas é a sequência que está sendo realizado, então no $Processo\ 0$, não podemos ter uma execução da $processNP$, por ser uma operação bloqueante, que só ira começar a receber os MPI_Send dos outros processadores, quando terminar sua execução. Mesmo sendo de grande utilidade, pois é um processador que fica ocioso, durante a execução.

Uma ideia que tivemos é criar utilizar Openmp e $2\ threads$, no $Processador\ 0$, que então poderíamos aproveitar o poder de processamento de mais um núcleo, que ficaria ocioso, então dividiríamos em $2\ sections$, uma para a realização da chamada da função de cálculo das senhas, e outra para receber os resultados, porém isso afetaria outras partes do código, como a divisão dos limites, onde deveria ser substituido o início do for , para a inclusão do $Processador\ 0$, além da inclusão de verificações na função $processNP$, que pode gerar um deadlock caso deixe pra retornar o valor encontrado para ela mesma, sendo terminada a execução ali mesma, pois o return torna a esperar no MPI_Recv.

Listing 1: Função Distribui

2.2 Demais Processadores

Seguindo o padrão de $mestre/escravo$ esse trecho de código representa os processos $escravos$, que apenas recebem os valores via MPI_Recv, neste caso alocando os valores dos seus limites e logo em seguida, chamando a função $processNP$.

2.3 Função processNP

A função $processNP$ resume-se no seu núcleo, sendo utilizado a forma de for concatenado, passível de otimizações, mas o tempo não permitiu.

Nesse caso cada loop representa um carácter para a iteração, então basicamente cada tamanho tem seus próprios loops para repetição, sendo criada uma String, para comparação, e se positiva segue o envio das mensagens para o $Processador\ 0$, sabendo que será enviado a String que foi encontrada, e o tamanho dela.

Na opção que utilizamos o Openmp, temos o if para definir que se for do $Processo\ 0$, exibe e aborta a execução do programa.

Listing 2: Núcleo de processamento

3 Experimentos

3.1 Cluster

4 Gráficos

4.1 Tempo

4.2 Discussão

4.3 Distribuição

4.4 Número de Processadores

André Thiago Borghi Couto atualmente cursa graduação em Engenharia da Computação (2016/1) na Universidade Federal do Espírito Santo. Possui conhecimento em desenvolvimento de software, banco de dados, projetos em Arduino, design gráfico e tem interesse nos temas de automação em Arduino, inteligência artificial, redes de computadores e processamento de imagens.