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 `strlen(caracteres) / np` 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 `strlen(caracteres) / (np - 1)`.

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.

```
int main(int argc, char *argv[]) {

    char *SENHA = argv[1];
    char *RESPOSTA;
    char caracteres[] = "abcdefghijklmnopqrstuvwxy⌋
      ↪  zABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";

    int np, rank;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    MPI_Status status;

    // ... restante do a ser código paralelizado
}
```

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 $escravos$.

- *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;

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, 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 `for(int i = 0; i < np; i++)`, para a inclusão do $Processador$ 0, além da inclusão de verificações `if(rank == 0) // exibe o resultado e aborta` 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.

```
if(rank == 0) {
    int **limites = (int **) calloc(np, sizeof(int
      ↪  *));
    for (int i = 0; i < np; i++)
        limites[i] = (int *) calloc(np,
          ↪  sizeof(int));
    distribui(limites, caracteres, np);
    for(int i = 1; i < np; i++){
        int limite[2];
        limite[0] = limites[i][0];
        limite[1] = limites[i][1];
        MPI_Send(limite, 2, MPI_INT, i, 0,
          ↪  MPI_COMM_WORLD);
    }
/*  #pragma omp parallel sections num_threads(2)
    {
    #pragma omp section
    {
        processNP(rank, limites[0], caracteres,
  ↪  SENHA);
    }
    #pragma omp section
    {
*/
    int numCache;

    MPI_Probe(MPI_ANY_SOURCE, 1, MPI_COMM_WORLD,
      ↪  &status);
    MPI_Get_count(&status, MPI_CHAR, &numCache);
    RESPOSTA = (char *) calloc(numCache,
      ↪  sizeof(char));

    MPI_Recv(RESPOSTA, numCache, MPI_CHAR,
      ↪  MPI_ANY_SOURCE, 1, MPI_COMM_WORLD,
      ↪  &status);
    cout << endl <<"O Processador " <<
      ↪  status.MPI_SOURCE << " achou a RESPOSTA"
      ↪  << endl << RESPOSTA << " : " << SENHA <<
      ↪  endl;
    MPI_Abort(MPI_COMM_WORLD, 1);
// }
// }
} else // continua nos Demais Processadores
```

```
int ind;
int **distribui(int **limites, char *c, int np){
    for (int /*i = 0*/ i = 1; i < np; i++){
        int ini, fim;
        if (i == np - 1){
            ini = ind;
            fim = strlen(c);
        } else {
            ini = ind;
            fim = ind + strlen(c) / (np - 1);
            ind += strlen(c) / (np - 1);
    }
    limites[i][0] = ini;
    limites[i][1] = fim;
    }
}
```

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 $processNP$.

```
} else {
    int *limite = (int *) calloc(2, sizeof(int));
    MPI_Recv(limite, 2, MPI_INT, 0, 0,
      ↪  MPI_COMM_WORLD, &status);
    processNP(rank, limite, caracteres, SENHA);
}
```

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.

```
for(int n = 0; n < tam; n++){
    s6[0] = caracteres[i];
    s6[1] = caracteres[j];
    s6[2] = caracteres[k];
    s6[3] = caracteres[l];
    s6[4] = caracteres[m];
    s6[5] = caracteres[n];
    string cript(sha1(s6));
    if(!cript.compare(SENHA)){
/*      if(rank == 0){
            cout << "O Processador rank " << rank
  ↪  << " achou a RESPOSTA" << endl << s5 << " : "
  ↪  << SENHA << endl;
            MPI_Abort(MPI_COMM_WORLD, 0);
        } else */
            MPI_Send(s6, 7, MPI_CHAR, 0, 1,
              ↪  MPI_COMM_WORLD);
    }
}
```

Listing 2: Núcleo de processamento

3 Experimentos

Os experimentos foram feitos no *Cluster* de computação do Ceunes, que por ser obrigatório, todos os resultados presentes deste trabalho foram obtidos por meio da submissão da execução dos programas no *Cluster*.

Utilizamos os padrões para envio, como o exemplo de $-np$ 32, que foi submetido para gerar estatísticas, e serem analisadas na próxima seção. Os experimentos envolveram a quebra de uma série de senhas, que foram criptografadas e submetidas para todos os $-np's$,

```
#!/bin/bash
#$ -cwd
#$ -j y
#$ -S /bin/bash
#$ -N Andre
#$ -pe orte 32
#$ -o output_par.txt
#$ -e errors_par.txt

env time -f "%C\n\tTempo de Execução:
 ↪  %e\n\tMemória usada: %M\n\tCPU usada: %P\n"
 ↪  -o SaidaTestes.txt -a mpirun -np 32 -quiet
 ↪  EP2 86f7e437faa5a7fce15d1ddcb9eaeaea377667b8
 ↪  >>SaidaTestes.txt
env time -f "%C\n\tTempo de Execução:
 ↪  %e\n\tMemória usada: %M\n\tCPU usada: %P\n"
 ↪  -o SaidaTestes.txt -a mpirun -np 32 -quiet
 ↪  EP2 6b0d31c0d563223024da45691584643ac78c96e8
 ↪  >>SaidaTestes.txt
env time -f "%C\n\tTempo de Execução:
 ↪  %e\n\tMemória usada: %M\n\tCPU usada: %P\n"
 ↪  -o SaidaTestes.txt -a mpirun -np 32 -quiet
 ↪  EP2 395df8f7c51f007019cb30201c49e884b46b92fa
 ↪  >>SaidaTestes.txt
```

O comando `env time -f -o -a` é um recurso do sistema operacional que tem por função exibir informações da tabela de processos da CPU, resumidamente este comando faz *-f* junto com uma String com as informações que deseja (ver $man\ time$), *-o* é o arquivo que vai ser salvo, e *-a* é o append, para adicionar ao final do arquivo.

```
O Processador 14 achou a Resposta
000000 : c984aed014aec7623a54f0591da07a85fd4b762d
mpirun -np 16 -quiet EP2
 ↪  c984aed014aec7623a54f0591da07a85fd4b762d
        Tempo de Execução: 4632.40
        Memória usada: 21760
        CPU usada: 1399%
```

3.1 Cluster

Para experimentos no *Cluster*, utilizamos um arquivo $run.sh$ para cada $-np$, assim as seguintes strings foram submetidas:

0, 00, 000, 0000, 00000, 000000, 012ab, 54321, 9, 99, 999, 9999, 99999, 999999, A, a, A1B2, A2019, aa, AA, AAA, aaa, aaaa, AAAA, aaaaa, AAAAA, aaaaaa, AAAAAA, abc12, AbCdE, admin, amor, ceunes, E, EE, EEE, EEEE, EEEEE, EEEEEE, etc, globo, jesus, love, m, M, mateus, mm, MM, mmm, MMM, mmmm, MMMM, mmmmm, MMMMM, mmmmmm, MMMMMM, money, ninja, pass, Red, senha, sexy, Z, z, ZZ, zz, zzz, ZZZ, ZZZZ, zzzz, ZZZZZ, zzzzz, zzzzzz, ZZZZZZ.

Em nossos experimentos obtivemos grandes resultados, com código paralelizado, incluímos novos testes para otimizar o código, para melhor proveito da estratégia, em que temos a divisão da primeira sequência de palavras, possibilitando a redução da complexidade do próximo nível em $n - 1$, ou seja, ao dividirmos o vetor de caracteres em 62, no primeiro nível, se levarmos em conta 5 níveis, apenas temos que calcular os 4 subsequentes, para implementações futuras, seria uma boa ideia aplicar essa técnica em mais de 1 nível, aumentando assim o paralelismo do código.

4 GRÁFICOS

4.1 Tempo

Levando em consideração o tempo, como nosso método utiliza formas de divisão sem saltos, ele proporciona a chegada mais rápida em chaves que podem estar no fim de qualquer ordem de caracteres, assim aumentando o desempenho relativo a sequência que é analisada, distribuindo mais opções à busca.

Em alguns casos tivemos problemas com o retorno dentro do Cluster, perdendo algumas das informações, e interferindo na análise desses dados, foi atribuído o valor 1.01 para os espaços vagos, assim alguns picos desnecessários estão aparecendo, principalmente em 3, que foi subdividido para cada senha, o desempenho com a quantidade de $-np$, mas mesmo assim, representando o desempenho esperado.

4.2 Discussão

Em nosso trabalho desenvolvemos o *divide and conquer*, no qual a divisão torna o trabalho mais fácil, em se tratando de programação paralela, isso é o básico, sendo que necessariamente precisamos de novas formas de relacionar processos e ganho de otimização em tempo, temos um grande desafio, onde a biblioteca escolhida contém peculiaridades novas e difíceis de assimilar, nas quais foi preciso algum tempo para a aplicação realmente funcionar e dar resultados, mesmo com isso, ainda poderíamos melhorar o código, mas a questão tempo, em um momento acadêmico conturbado contribuiu negativamente para a saída de um trabalho exemplar.

Então levando em conta os resultados dos gráficos, temos visto que em determinadas caracteres de início temos uma velocidade alta de resposta, logo porém caso a distribuição seja errônea, o resultado pode torna-se o último de uma fila, que está na distribuição, sendo que uma chamada de $-np$ menor que a atual, seja mais eficaz, porém isto se torna fora de mão, pois o que interessa é encontrar a senha apenas uma vez, qualquer modificação no código para direcionar a resposta para uma única chave, pode tornar outras mais difíceis, chegando a conclusão que em nosso código uma sequência de $-np\ (strlen(caracteres))$ é a opção mais recomendada, pela divisão em primeiro nível tornar-se mais eficiente.

5 ANEXOS

		Tempo em Segs			
SHA1	Senha	1	8	16	32
8cb2237d0679ca88db6464eac60da96345513964	12345	7975.25	784.24	160.25	160.28
7b151de317f2547df39e1a1ff2850a6abfa6128f	A1B2C	3635.25	376.64	268.34	87.79
230cdfcdb6f2aa33b7acf19edaae5a216a14155f	123ZZ	6656.25	616.92	156.48	160.71
b2cdbf0601d8ae90d3cda1c978566ace86c4eac0	DCEL	55.81	8.39	3.30	3.58
a045b7efa463c6ed195c644163f4168952fbd34a	99999	7585.25	1270.85	708.76	166.1
d27e086d60a993d203717509c46a6752daccc967	CEUNES	165944.94	35722.63	14395.43	2869.29

Tabela 1
Tabela de senhas Descriptografadas

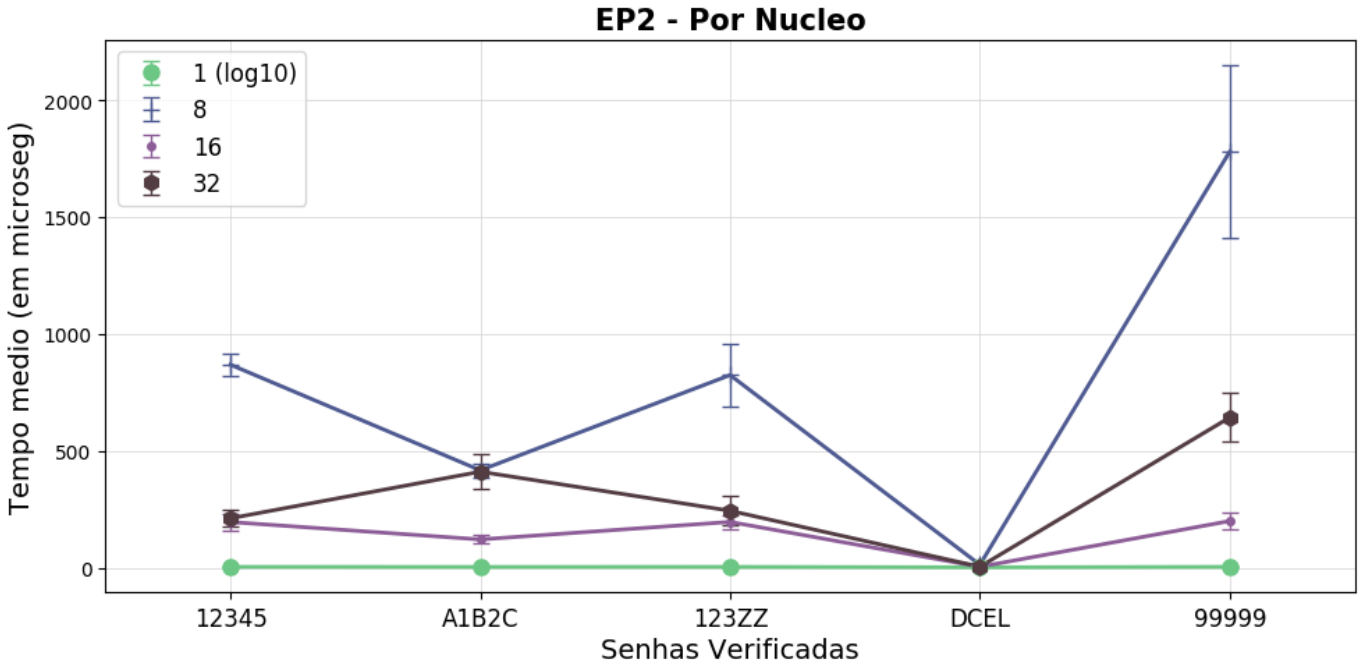

Figura 1. Gráfico de execução separado por núcleo

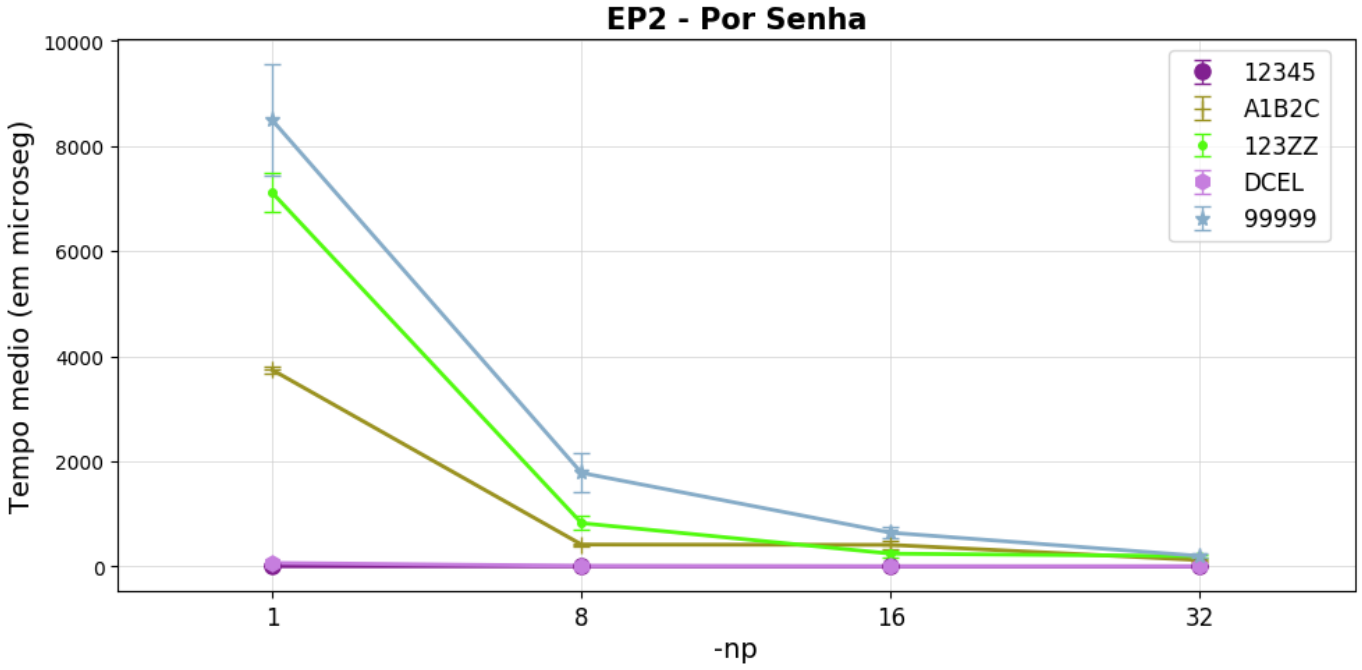

Figura 2. Gráfico de execução separado por senha

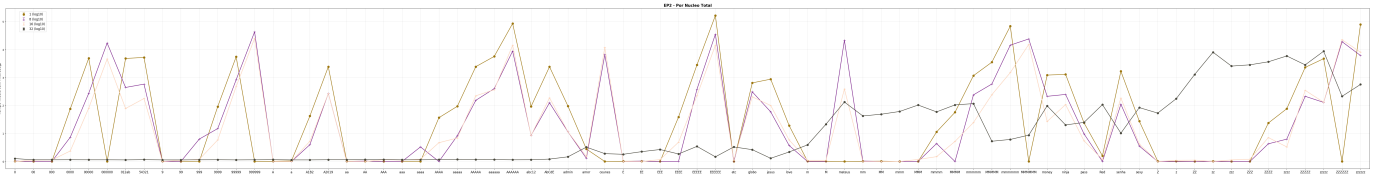

Figura 3. Gráfico de execução separado por núcleo com todos os testes, disponível aqui

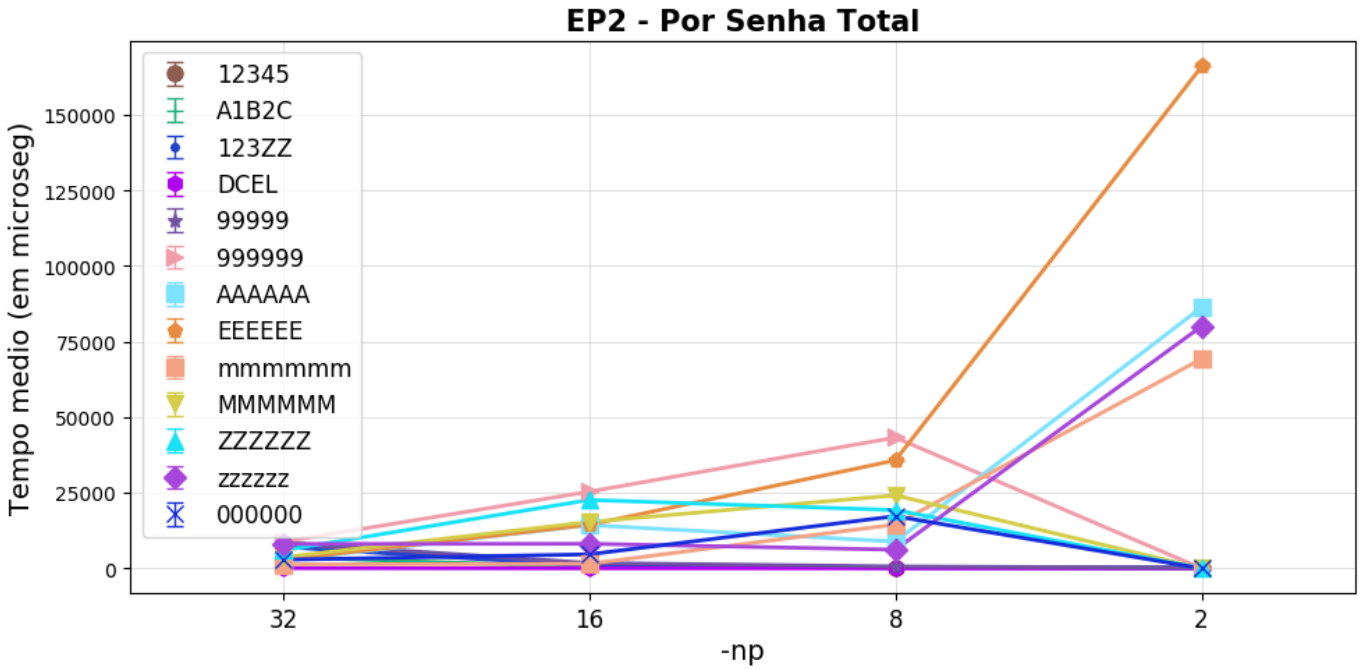

Figura 4. Gráfico de execução separado por senha com todos os testes