UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO

Curso de Ciência da Computação

**Processamento Paralelo e Distribuído**

Segundo Trabalho

**Alunos:** Higor Borjaille Soneguetti

Bernardo Lucas Sunderhus

**Professor:** João Paulo Andrade Almeida

Vitória - ES

27 de junho de 2016

Sumário

[1 - INTRODUÇÃO 4](#_Toc454740430)

[1.1 – Arquitetura Mestre-Escravo 4](#_Toc454740431)

[1.2 – Ataque de dicionário em Mensagem Criptografada 5](#_Toc454740432)

[1.3 – Objetivo 5](#_Toc454740433)

[2 – IMPLEMENTAÇÃO 6](#_Toc454740434)

[3 - MANUAL DE EXECUÇÃO 8](#_Toc454740435)

[4 - ANÁLISE DO DESEMPENHO 9](#_Toc454740436)

[4.1 Análise do desempenho no tempo em paralelo em uma máquina Dual Core 10](#_Toc454740437)

[4.2 Análise do desempenho no tempo para uma solução distribuída vs serial 11](#_Toc454740438)

[4.3 Análise do speedup e eficiência para a solução paralela e distribuída 13](#_Toc454740439)

[5 – CONCLUSÃO 16](#_Toc454740440)

Lista de Figuras

Figura 1 - Diferenças nos tempos de execução sequenciais e paralelos. 10

Figura 2 - Diferença nos overheads entre dois à cinco escravos em uma máquina com dois núcleos de processamentos 10

Figura 3 - Diferença no tempo para uma solução distribuída com dois à cinco escravos 11

Figura 4 - Diferença nos overheads para uma solução distribuida com dois à cinco escravos 12

Figura 5 - Speedups para solução paralela com dois à cinco escravos em uma máquina com dois núcleos 13

Figura 6 - Speedups para uma solução distribuida com dois à cinco escravos em máquinas distintas 13

Figura 7 - Eficiência para uma solução paralela com dois à cinco escravos em uma máquina com dois núcleos 14

Figura 8 - Eficiência para uma solução distribuida com dois à cinco escravos em máquinas distintas 14

# 1 - INTRODUÇÃO

Este trabalho foi desenvolvido para colocar em prática os conceitos de programação paralela e distribuída e ser realizado um Ataque de Dicionário em Mensagem Criptografada (sessão 1.2). Na implementação do trabalho foi explorada a arquitetura mestre-escravo com tolerância à falha que está detalhada na sessão 1.1 deste relatório.

Neste relatório serão apresentados testes distribuídos em duas ou mais máquinas ou dois ou mais diferentes processos em uma única máquina, utilizados para comparar o tempo de execução, speedup, eficiência e overhead para diferentes tamanhos de mensagens com seus respectivos testes em serial (sem distribuição).

## 1.1 – Arquitetura Mestre-Escravo

Mestre-escravo é um modelo de comunicação onde um processo “mestre” tem controle unidirecional sobre um ou mais processos, normalmente presentes em um ambiente distribuído.

Em outras palavras a arquitetura mestre-escravo é uma configuração usada para resolver o balanço de carga entre servidores. O mestre agiria como um balanceador que emite o que cada servidor “escravo” deve fazer, garantindo assim distribuição e paralelismo.

Esta arquitetura é recomendada quando há varias processos independentes a serem executados e o tempo de execução é de extrema importância.

Existe um acoplamento no tempo e espaço com relação ao mestre e seus respectivos escravos, ou seja, o mestre necessita estar ativo juntamente com seus escravos e os mesmos se conhecem, porém para que uma certa tolerância a falha seja inserida, algumas ações tiveram que ser tomadas.

Por exemplo:

Para que o escravo saiba que o mestre está ativo, o mesmo lhe oferece uma interface de checkpoint, para que ele de tempos em tempos se re-registre e caso o re-registro falhe o escravo busca no registro de nomes o novo endereço do mestre.

O mestre faz uma verificação em cima dos checkpoints realizados pelos escravos e se algum deles estiver inativo por mais de um determinado tempo, o mesmo desregistra o escravo tomado como inativo.

## 1.2 – Ataque de dicionário em Mensagem Criptografada

O ataque consiste em procurar chaves possíveis dentro de um dicionário de 80.368 palavras em inglês para desencriptar o texto utilizando o algoritmo Blowfish.

Blowfish é uma cifra de blocos simétrica que encripta dados em blocos de tamanho 8 bytes e possui uma chave de tamanho variável entre 32 a 448 bits, com subchaves previamente computadas, cada subchave é uma hash simples da chave, não possui estruturas lineares, possui uma implementação de simples entendimento. O algoritmo faz uso da rede de Feistel de 16 iterações onde cada iteração consiste em uma permutação da chave-dependente, e uma substituição da chave-dado dependente. A rede de Feistel é uma estrutura simétrica usada na construção de cifras de blocos responsável por transformar qualquer função em uma permutação, mapeando uma string de entrada para uma string de saída.

## 1.3 – Objetivo

O principal objetivo do trabalho foi realizar a análise do tempo de resposta observado pelo cliente para diferentes tamanhos de mensagem, onde são realizadas tentativas para a decriptar a mensagem com todas as palavras do dicionário utilizando a arquitetura distribuída mestre-escravo em comparação com os mesmos feitos em serial.

Na distribuição os escravos se registram no mestre previamente. O mestre recebe uma mensagem encriptada enviada pelo cliente e separa os índices do dicionário para cada escravo, sendo que o dicionário já foi previamente carregado nos escravos, então a mensagem é distribuída para cada escravo que tenta fazer a decriptação.

# 2 – IMPLEMENTAÇÃO

A classe **MasterBH**, que implementa a interface **Master**, é incumbida tanto de gerenciar o registro e a exclusão de escravos como também de controlar o fluxo de divisão dos ataques requisitados pelo cliente.

Ao ser invocado pelo cliente com a operação master.attack(ciphertext, knowntext) para processar um ataque de dicionário ao texto cifrado, o mestre deve subdividir a tarefa entre cada escravo. Após dividir igualmente espaçado o dicionário é executado slave.startSubAttack(cipherText, knownText, initialWordIndex, finalWordIndex, callbackInterface) de forma não bloqueante. Logo em seguida uma thread é invocada para vigiar cada um dos escravos, a espera de checkpoints que confirmem que o escravo não está ocioso.

Após iniciar a thread o mestre entra em estado de espera até que seja acordado então ao receber a confirmação de que todos os escravos terminaram.

O mestre também é responsável por reconhecer quando um escravo realiza um checkpoint e também quando o escravo acha um **Guess**.

Ao receber um Guess o mestre deve adicionar o guess a lista de guesses a serem retornados para o cliente.

Ao receber um checkpoint o mestre atualiza as informações do escravo, mantendo assim o seu atual posicionamento no dicionário e também atualizando quando foi a última vez que o checkpoint ocorreu, por último é verificado se o dado checkpoint indica o final do ataque, caso indique o mestre é então acordado e a thread de vigia é destruida e a função de ataque retorna ao cliente.

A classe **SlaveAttack** é criada para garantir uma granularidade maior de encapsulamento ao processamento das atividades dos escravos. Ao ser iniciado um ataque, o mestre cria uma instância da classe e então a adiciona a uma lista global de ataques, mantendo assim referência a todos os ataques vigentes.

attack = new SlaveAttack(slave, initialWordIndex, finalWordIndex, this);

attacks.add(attack);

O objetivo dessa classe, além de manter o encapsulamento, é garantir consistente de informações de acesso, como a última vez em que o escravo deu o seu checkpoint e em que índice o escravo está.

A classe **SlaveBH** que implementa a interface **Slave**, define o escravo utilizado pelo mestre para realização do ataque particionado de dicionário.

O escravo deve por definição procurar pelo nome *mestre* na tabela de registro no ***host*** informado em tempo de execução.

slave**.**slaveManager **=** (**SlaveManager**) registry**.**lookup("mestre");

Caso não encontrado o escravo deve tentar de novo a cada 30 segundos, também é necessário que o escravo pergunte de tempos em tempos se o mestre está disponível pelo registro, caso não esteja volte a procurar o mestre a cada 30 segundos.

slave**.**reRegistrationHook(host);

A principal atividade de um escravo é realizar a operação startSubAttack(ciphertext, knowntext, initialwordindex, finalwordindex,cb) de forma não bloqueante. Para realizar tal operação o escravo cria uma **Thread** que irá realiza o ataque e então à inicializa, retornando assim ao mestre o fim da função.

Ao mesmo tempo é também inicializada uma **Thread** que roda de tempos em tempos (10 segundos definido pela implementação vigente) uma função que envia um checkpoint ao mestre, informando em que índice aquele ataque se encontra.

checkpointScheduler**.**scheduleAtFixedRate(automaticCheckpoint, 10, 10, **SECONDS**);

Ao final do ataque outro checkpoint é enviado indicando assim o final.

O código utilizado para medir o tempo de execução do teste serial está mostrado abaixo:

startTime = System.nanoTime();  
 Guess[] guesses = master.attack(cipherText, knownString);  
 elapsedTime = System.nanoTime() - startTime;

# 3 - MANUAL DE EXECUÇÃO

Para que se possa executar o Sistema é necessário que o Java esteja instalado corretamente.

1. Entrar no diretório desejado, neste caso “~/TrabalhoPP2”
2. Gerar os .class desejados: javac \*.java
3. Depois dos .class gerados podemos executar o Sistema na seguinte ordem:

Mestre:

java -Djava.rmi.server.hostname=IPDoMestre MasterBH

Escravo(s):

java -Djava.rmi.server.hostname=IPDoEscravo SlaveBH IPDoMestre NomeDoSlave

Cliente:

java Client IPDoMestre arquivoCifrado TextoConhecido

# 4 - ANÁLISE DO DESEMPENHO

Durante esta sessão serão discutidos alguns testes realizados para realizar a comparação da execução serial vs paralela e distribuída para o ataque de dicionário em mensagem criptografada. O objeto de estudo desta análise é o speedup e eficiência das soluções paralela e distribuída. Portanto, foram considerados mensagens de tamanho variando entre [], o tempo de resposta paralelo e distribuído e sequencial.

Para a solução proposta foi explorada a decomposição de dados de saída, pois cada elemento da saída pode ser computado independentemente. A decomposição é realizada através da atribuição de uma tarefa ao cálculo de cada subconjunto da partição dos dados e esta decomposição só pode ser utilizada porque o dicionário está copiado em cada máquina Slave.

No sistema distribuído com o uso da arquitetura mestre-escravo o mestre foi executado na mesma máquina que o serviço de nomes, o cliente e os escravos foram executados em diferentes máquinas. Todas as máquinas utilizadas possuem a seguinte arquitetura:

* Processador AMD Athlon(TM) Dual Core Processor 5000 X2
* Memória 4GB
* Rede de 100Mb/seg
* Sistema Operacional Linux Mint 17.1 Cinnamon 2.4.6 64 bit kernel  3.13.0­37­generic

Os testes foram realizados no Laboratório de Graduação (LabGrad) no prédio CT9 da Universidade Federal do Espírito Santo.

## 4.1 Análise do desempenho no tempo em paralelo em uma máquina Dual Core

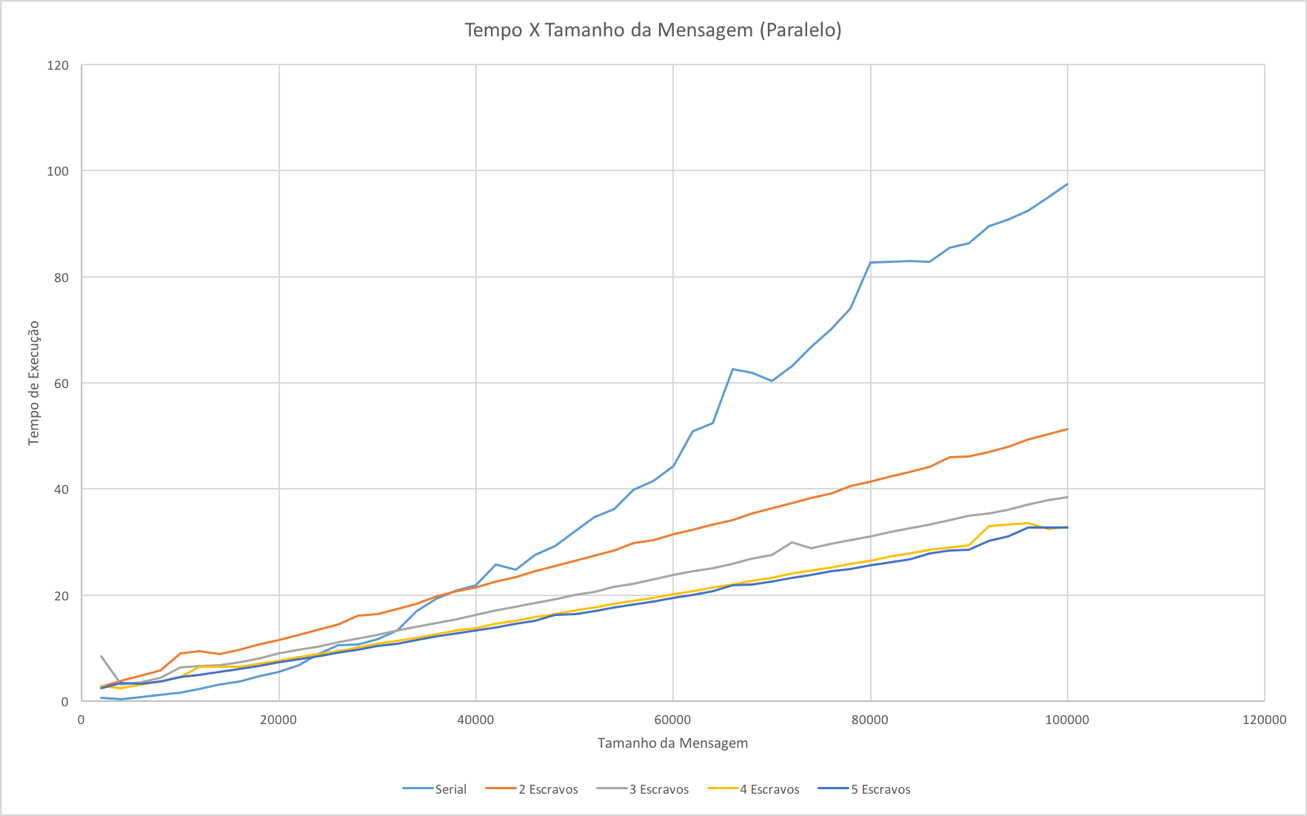


Figura 1 - Diferenças nos tempos de execução sequenciais e paralelos.

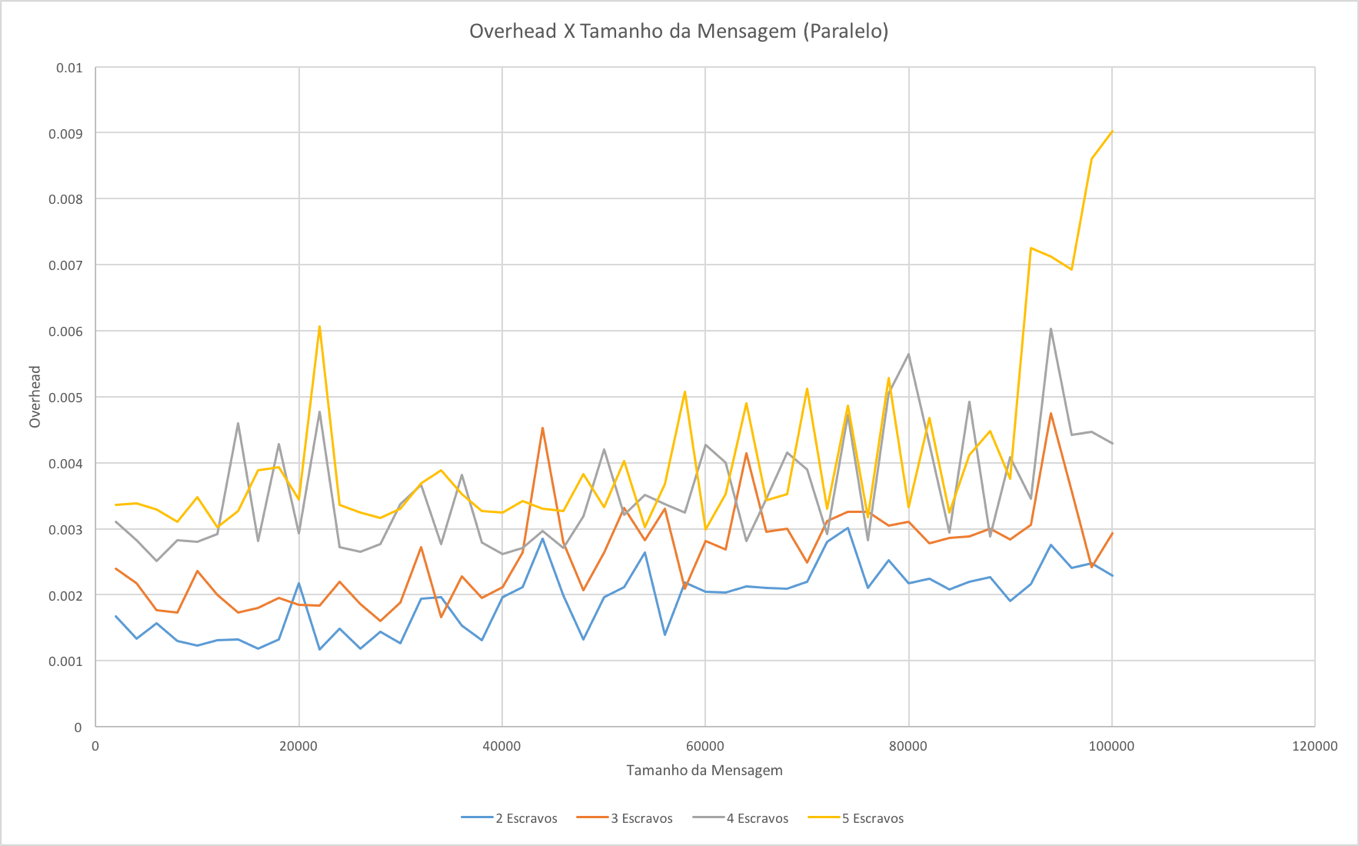


Figura 2 - Diferença nos overheads entre dois à cinco escravos em uma máquina com dois núcleos de processamentos

Podemos observar que a cada escravo adicionado obtemos um tempo de resposta melhor do que o anterior mesmo levando em consideração o acréscimo de um overhead que tem crescimento linear de acordo com cada escravo adicionado. O limite imposto pela máquina faz com que tenhamos uma quantidade máxima de escravos que podemos adicionar para a solução antes que o sistema operacional faça com que dois ou mais processos concorram por um núcleo de processamento, neste caso quatro escravos é a quantidade máxima, podemos observar o quão próximo a curva da execução com 4 escravos está da curva de execução com 5 escravos.

## 4.2 Análise do desempenho no tempo para uma solução distribuída vs serial

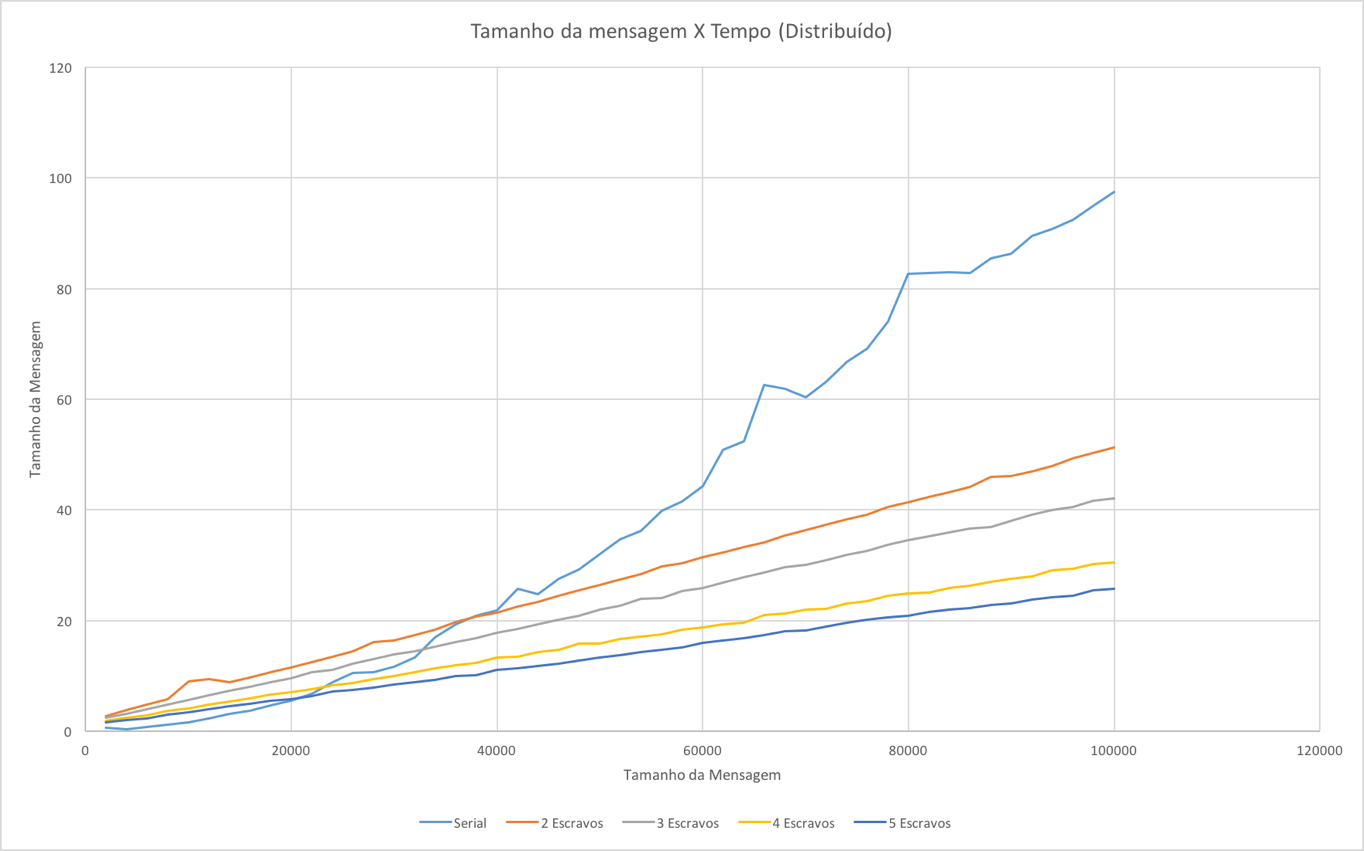


Figura 3 - Diferença no tempo para uma solução distribuída com dois à cinco escravos

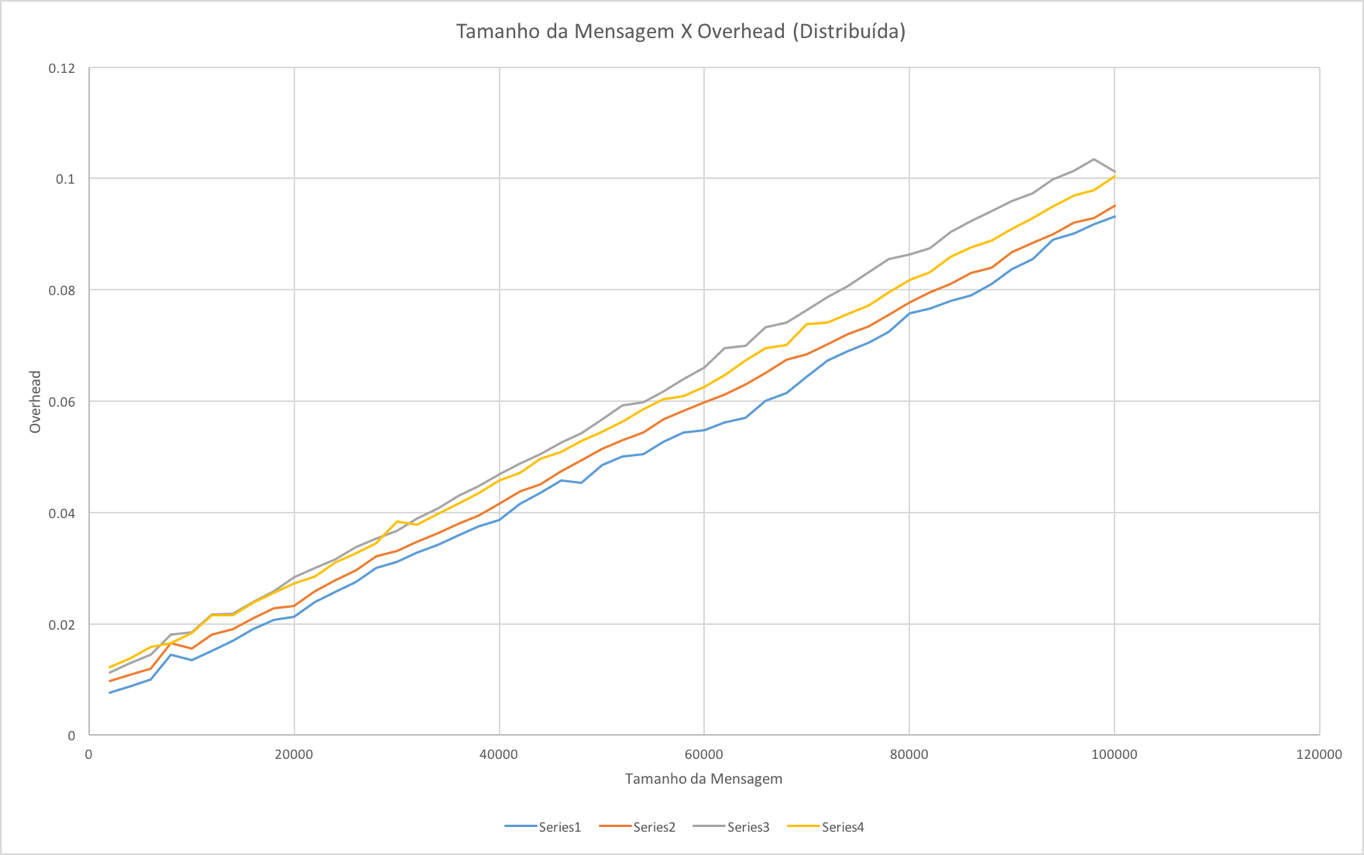


Figura 4 - Diferença nos overheads para uma solução distribuida com dois à cinco escravos

Podemos observar no gráfico da figura 3 que a solução serial começa a ter um desempenho no tempo melhor para a solução distribuída, quando o tamanho da mensagem passa de XXX KB e essa diferença de tempo pode ser atribuída a paralelização do trabalho total.

Diferente de uma solução paralela que tem um ganho limitado devido ao número de cores disponíveis da maquina em questão. Na solução distribuída observamos que a cada escravo adicionado podemos ter um ganho maior no tempo de execução, pois não existe uma limitação local e somente uma limitação em relação à porcentagem de execução do código que é não paralelizável, portanto hipoteticamente se não existisse um crescimento do overhead descrito pelo gráfico da figura 4, que demonstra o aumento do overhead de comunicação para cada máquina adicionada ao sistema, a medida que cada novo escravo fosse adicionado, poderíamos observar uma redução ainda maior do tempo, porém como descrito pelo gráfico da figura 3, o ganho no tempo já não é tão grande com a adição do quinto escravo, então concluindo que o ganho máximo de desempenho no tempo é limitado pelo crescimento do overhead.

## 4.3 Análise do speedup e eficiência para a solução paralela e distribuída

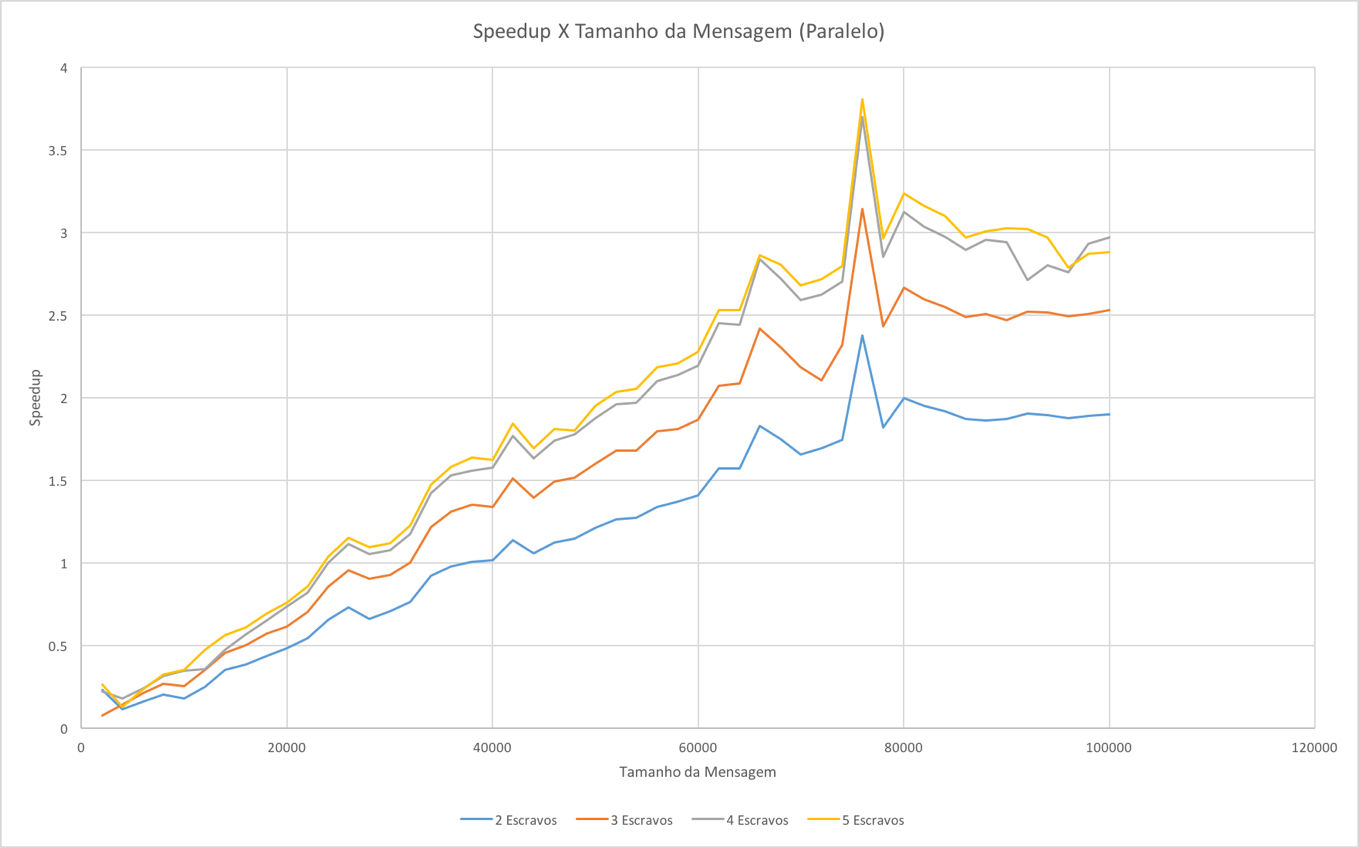


Figura 5 - Speedups para solução paralela com dois à cinco escravos em uma máquina com dois núcleos

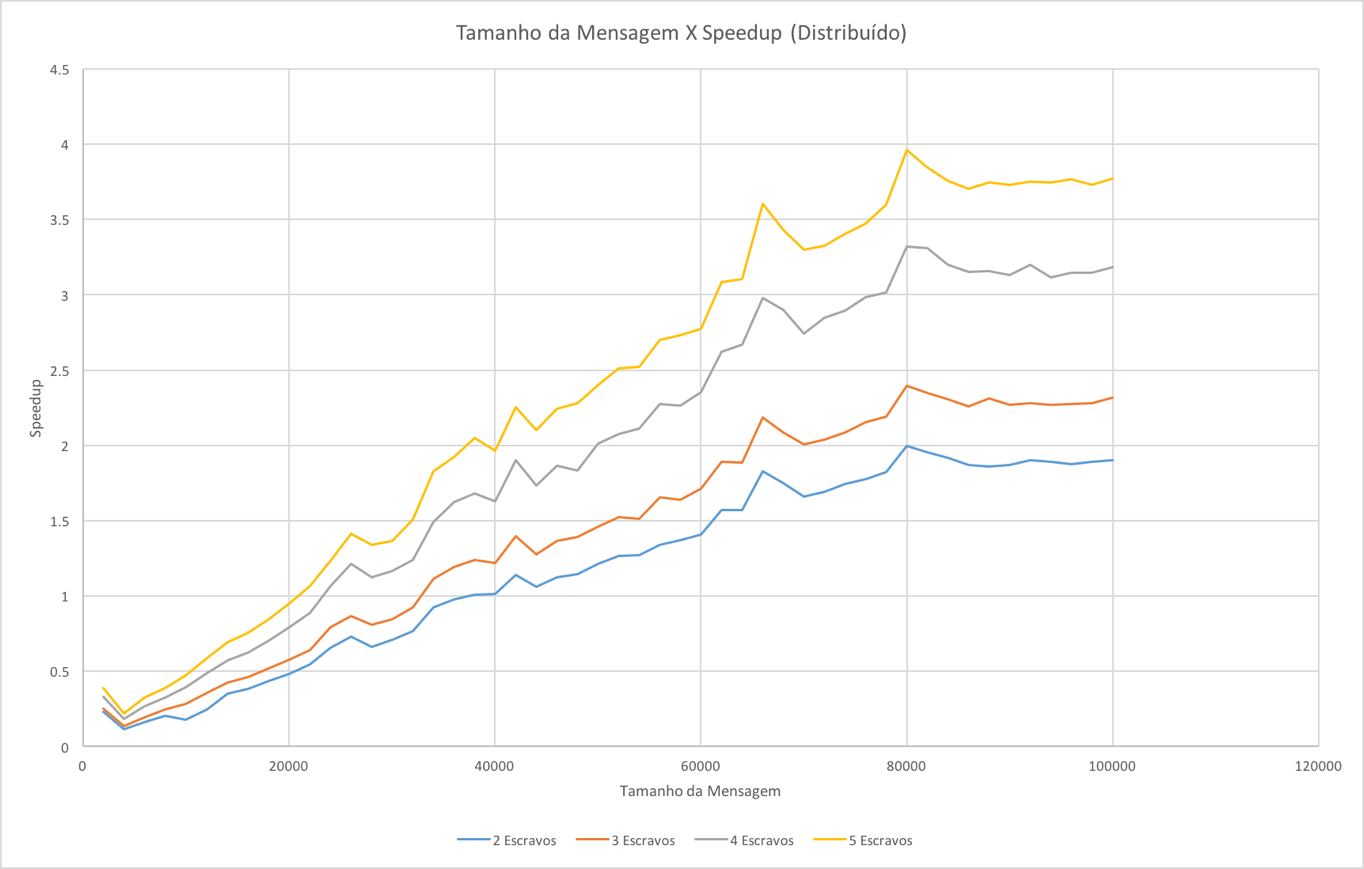


Figura 6 - Speedups para uma solução distribuida com dois à cinco escravos em máquinas distintas

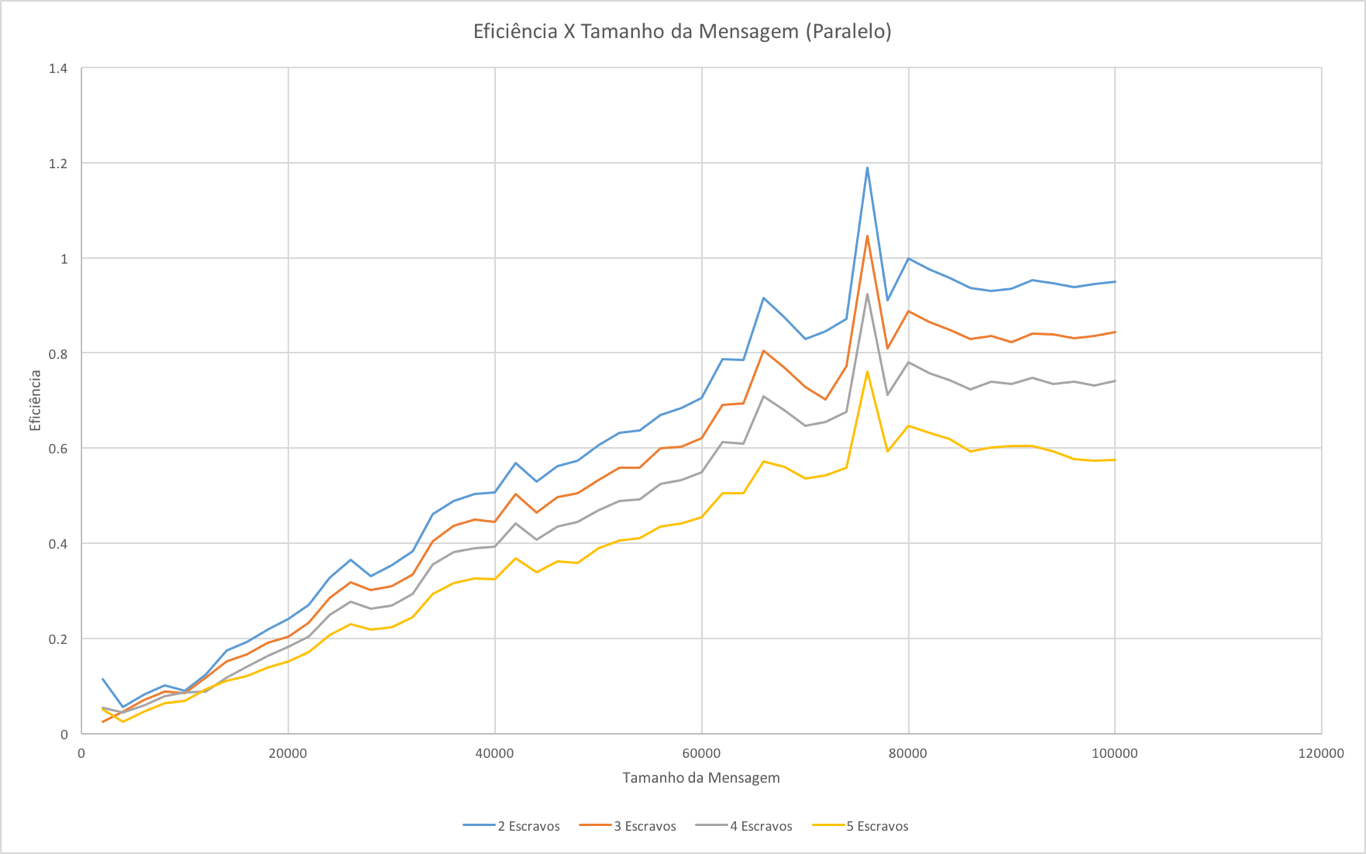


Figura 7 - Eficiência para uma solução paralela com dois à cinco escravos em uma máquina com dois núcleos

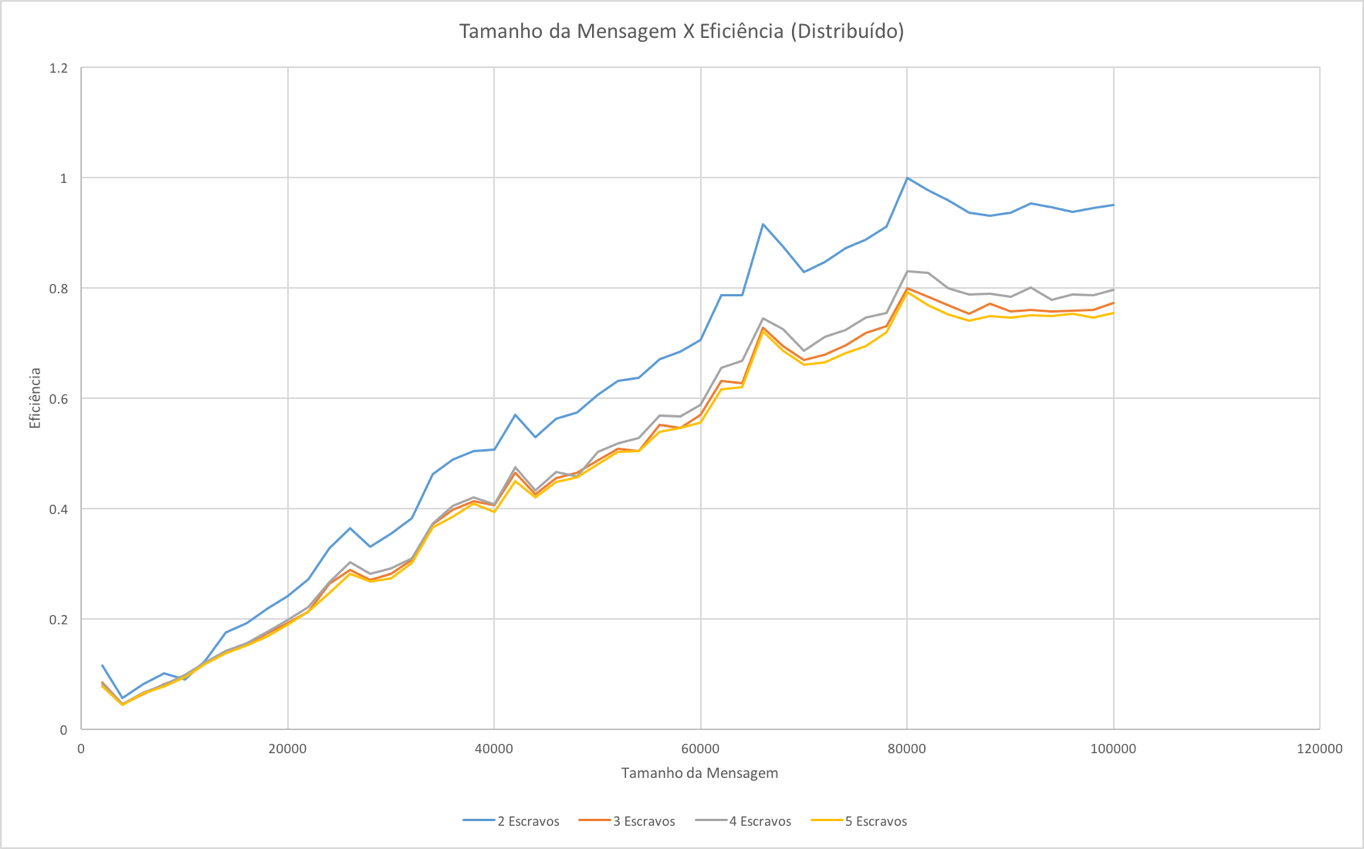


Figura 8 - Eficiência para uma solução distribuida com dois à cinco escravos em máquinas distintas

Observando o caso paralelo, comparando os gráficos das figuras 5 e 7, podemos concluir que o speedup também é limitado pela quantidade de núcleos de processamento disponíveis pela máquina, não permitindo que o speedup seja maior que a quantidade de núcleos de processamento. A eficiência por um outro lado cai a cada escravo adicionado, pois como a diferença dos speedups não é muito grande podemos dizer que estamos utilizando de um esforço computacional maior, para pouquíssimo ganho no desempenho.

Com o speedup da solução distribuída mostrado na figura 6, podemos ver que diferente da execução paralela, a execução distribuída com 5 escravos obteve um speedup maior, porque na solução distribuída não fica limitada a capacidade de processamento da máquina para determinar a quantidade de escravos. Era esperado, já que o speedup aumentasse dado que grande parte da execução total é paralelizável, e quanto mais escravos adicionarmos melhor seria o speedup, porém também esta solução é limitada de acordo com a porção não paralelizável do código e ao overhead.

# 5 – CONCLUSÃO

Neste relatório foi discutido uma solução paralela e distribuída para o problema de um ataque de dicionário em mensagem criptografada. Durante a análise de desempenho tecemos alguns comentários a respeito do tempo de execução, speedup, eficiência e overhead.

Para decriptar uma mensagem criptografada é uma tarefa exaustiva onde temos que explorar todas as chaves possíveis para a decriptação, portanto utilizamos de uma decomposição de dados de saída para que possamos ter um overhead menor assumindo que o trafico de dados será menor pois o dicionário neste caso está espelhado.

Este problema é melhor solucionado pela distribuição apesar de suas limitações de acordo com a paralelização do código, pois a quantidade de dados a ser analisada e o esforço computacional são maiores, ao contrario da verificação de integridade onde a solução serial é bem mais rápida que a solução distribuída.

Obtivemos um speedup maior que um para todas as soluções propostas com dois à cinco escravos a partir do tamanho XXX KB de mensagem, portanto podemos concluir que não necessariamente sempre a solução distribuída é melhor que a solução serial, porém neste problema em específico podemos obter um speedup maior que um por causa das características do mesmo.