UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO

CIÊNCIA DA COMPUTAÇÃO

PROCESSAMENTO PARALELO E DISTRIBUÍDO

MATEUS ALMEIDA DE OLIVEIRA

THIAGO MARTINHO DA COSTA

IMPLEMENTAÇÂO DA ARQUITETURA MESTRE-ESCRAVO EM UM SISTEMA PARALELO E DISTRIBUÍDO EM JAVA

VITÓRIA/ES

2015

1. Introdução

O objetivo deste trabalho é aplicar de forma prática os conceitos de programação paralela e distribuída estuda em sala. Para tal objetivo, foi utilizado o *middleware* JAVA RMI e a arquitetura mestre/escravo para sistemas paralelos e distribuídos. Foram realizados experimentos com diferentes quantidades de computadores para avaliar o paralelismo da ordenação de vetores de tamanhos variados.

A arquitetura mestre/escravo foi implementado na linguagem JAVA e foram feitos vários testes, com tamanhos de vetores variando de 1 até 1000000, em um sistema “distributível” (um sistema pronto para ser executado de forma distribuída, mas que ainda é processado todo na mesma máquina), um sistema distribuído de fato e também uma forma sequencial de execução.

2. O problema

O problema a ser resolvido utilizando um sistema paralelo e distribuído era o de ordenar um vetor de tamanho *n* utilizando o método de ordenação *MergeSort* que tem tempo de execução médio aproximado de *n*lg(n).*

A execução do conjunto consiste em um programa cliente que faz uma requisição ao mestre, de ordenar um vetor conhecido pelo cliente. O mestre por sua vez divide esse vetor em *k* subvetores de tamanhos iguais ou próximos da igualdade, onde *k* representa a quantidade de escravos registrados no mestre no momento em que a requisição do cliente chegou ao mestre. Depois de dividir todas as tarefas, o mestre inicia *k threads* para que cada escravo possa ordenar a sua parte do vetor de forma paralela (cada processo escravo rodando em um computador diferente) e ao final de todas as *threads* todos os vetores ordenados pelos escravos são recuperados e os dados são unificados através de um *Merge* efetuado pelo mestre. Por fim, o mestre retorna ao cliente o vetor ordenado como resultado da computação realizada.

2.1. A Implementação e Execução

Para executar a computação requisitada, foi necessária a execução de alguns comandos prévios, como por exemplo, iniciar o *Registry* em alguma máquina. A seguir está descrito todos os comandos executados antes de o cliente efetuar qualquer requisição.

Em uma máquina qualquer:

1) rmiregistry
2) java –Djava.rmi.server.hostname=xxx.xxx.xxx.xxx MestreImpl

Para cada escravo, ou seja, em máquinas diferentes:

3) java –Djava.rmi.server.hostname=xxx.xxx.xxx.xxx EscravoImpl yyy.yyy.yyy.yyy

Onde: xxx.xxx.xxx.xxx = IP local e yyy.yyy.yyy.yyy = IP da máquina em que o

O mestre foi representado através de uma interface chamada “Mestre” que tem como implementação uma classe chamada “MestreImpl”. Tal classe também implementa uma outra interface chamada “Ordenar” que tem o método de ordenar o vetor. Logo, foi necessário instalar na máquina que executaria o mestre os seguintes arquivos: *Mestre.class, MestreImpl.class, Ordenar.class, Escravo.class e GeradorID.class (*que é uma classe que faz o controle de Ids que serão associados a escravos). Como o mestre possuí uma referência de todos os escravos associados em si, foi necessário também ter a interface chamada “Escravo”.

Com os dois primeiros comandos, iniciamos o *registry* em uma máquina e também executamos o mestre na mesma máquina. A propriedade *Djava.rmi.server.hostname* representa o nome do *host* que deve ser associado com *stubs* remotos para objetos criados localmente, de modo a permitir que clientes invoquem métodos do objeto remoto. Com isso, foi necessário inserir o IP de rede da máquina em questão, como forma de deixar disponível o objeto remoto a ser acessado pelo IP. Após a execução deste comando, o mestre fica de prontidão para receber escravos e/ou requisições de clientes.

Já os escravos foram representados através de uma interface chamada “Escravo” que tem como implementação uma classe chamada “EscravoImpl”. Assim como o mestre, o escravo também implementa a interface “Ordenar” e possui a mesma chamada de método para ordenar vetores. Logo foi necessário instalar na máquina que executaria o escravo os seguintes arquivos: *Escravo.class, EscravoImpl.class e Ordenar.class*.

Já para o cliente, o seguinte comando foi executado:

Em uma máquina qualquer:

1) java –Djava.rmi.server.hostname=xxx.xxx.xxx.xxx Cliente NomeArquivo yyy.yyy.yyy.yyy

onde: NomeArquivo representar o nome do arquivo de entrada que contem em cada linha um tamanho do vetor a ser ordenado e yyy.yyy.yyy.yyy = IP da máquina em que o *registry* foi iniciado.

Para poder executar o cliente, os seguintes arquivos foram instalados na máquina: *Cliente.class e Mestre.class.*

A seguir, segue a definição das interfaces Mestre, Escravo e Ordenar.

```
import java.rmi.RemoteException;

public interface Mestre extends java.rmi.Remote, Ordenar {

    /**
     * Método usado para registro de um novo escravo no mestre
     *
     * @param escravo - o escravo que deseja ser adicionado ao mestre
     * @throws java.rmi.RemoteException
     */
    public void registraEscravo(Escravo escravo) throws
RemoteException;

    /**
     * Método usado para retirar um escravo do mestre
     *
     * @param idEscravo - id do escravo que deseja retirar
     * @throws java.rmi.RemoteException
     */
    public void retirarEscravo(int idEscravo) throws RemoteException;

    /**
     * Método que recupera a quantidade de escravos associados ao
mestre
     *
     * @return
     * @throws java.rmi.RemoteException
     */
    public int getQuantidadeEscravos() throws RemoteException;
}
```

```java
import java.rmi.RemoteException;

public interface Escravo extends Ordenar {

    /**
     * Método utilizado para recuperar o ID do escravo registrado pelo
mestre
     *
     * @return a lista de inteiros ordenada
     * @throws java.rmi.RemoteException
     */
    public int getId() throws RemoteException;

    /**
     * Método utilizado para atribuir ao escravo um ID
     *
     * @param id
     * @throws java.rmi.RemoteException
     */
    public void setId(int id) throws RemoteException;

    /**
     * Método utilizado para o mestre terminar o escravo quando
necessário
     *
     * @throws java.rmi.RemoteException
     */
    public void terminarEscravo() throws RemoteException;
}
```

```
import java.rmi.RemoteException;
import java.util.List;

public interface Ordenar extends java.rmi.Remote {

    /**
     * Método remoto que recebe a chamada do cliente para ordernar uma
lista de
     * inteiros
     *
     * @param numeros - lista de inteiros que deseja-se ordernar
     * @return a lista de inteiros ordenada
     * @throws java.rmi.RemoteException
     */
    public List<Integer> ordenarVetor(List<Integer> numeros) throws
RemoteException;

    /**
     * Método remoto que recebe a chamada do cliente para calcular o
overhead de
     * comunicação das chamadas remotas. A lista não é ordenada.
     *
     * @param numeros - lista de inteiros que deseja-se ordernar
     * @return a lista de inteiros passada como parâmetro
     * @throws java.rmi.RemoteException
     */
    public List<Integer> calcularOverhead(List<Integer> numeros)
throws RemoteException;

}
```

Pode-se observar que todas as interfaces estendem, diretamente ou indiretamente, a interface que permite ao objeto ficar disponível remotamente. Tal interface é a *java.rmi.Remote* ou simplesmente *Remote*. A interface “Escravo” herda da interface “Ordenar” e o mesmo vale para a interface “Mestre”. Além disso, tanto a interface Mestre quanto “Escravo”, possuem outros métodos como por exemplo o de cálculo de overhead.

Já para o caso sequencial, foi considerado que o próprio cliente é responsável por todas as etapas do processamento de ordenação do vetor.

Abaixo segue o trecho de código que representa a coleta do tempo gasto e a implementação do *MergeSort*.

```
/**
     * Método usado para ordenar a lista de inteiros de forma
sequencial
     *
     * @param writer - buffer para escrita em um arquivo de saída
     * @param numeros - lista
     */
    public static void versaoSequencial(PrintWriter writer,
List<Integer> numeros) {
        long nanoTime = System.nanoTime();
        Collections.sort(numeros);
        double nanoTime2 = System.nanoTime();
        double nanoTime3 = (nanoTime2 - nanoTime) / 1000000000.0;
}
```

O método utilizado para ordenar a lista no caso sequencial vem da interface Java *Collections*. O método dessa interface utilizado é o *sort*, que é um *MergeSort* modificado que ordena uma lista qualquer de inteiros em um tempo *n*lg(n)* (*https://docs.oracle.com/javase/6/docs/api/java/util/Collections.html*).

3. Experimentos

O objetivo dos experimentos era calcular o tempo de resposta e o *speed-up* das soluções. Na análise paralela e distribuída o mestre era executado na mesma máquina que o servidor de nomes, o cliente e cada escravo eram executados em máquinas diferentes. Na análise paralela e local todos os elementos do sistema rodavam na mesma máquina. As máquinas utilizadas são semelhantes e a configuração era:

Intel Core i5-2400 3.10GHz, x4
4GB RAM
Linux Mint 17.1 64-bit, kernel 3.13.0-37-generic

Os testes foram feitos no Laboratório de Graduação (LabGrad) do departamento de informática da Universidade Federal do Espírito Santo.

Abaixo será apresentado os testes realizados e seus resultados. As imagens apresentadas nessa sessão apresentação o resultado do tempo em segundos (eixo y) pelo tamanho do vetor (eixo x).

3.1. Solução Paralela e Distribuída

Podemos observar pela figura 1 que o tempo de execução sequencial em uma máquina é bem menor que o tempo feito pela solução distribuída. A diferença de tempo pode ser atribuída ao tempo de comunicação do middleware e também ao tempo de comunicação utilizando a rede local.

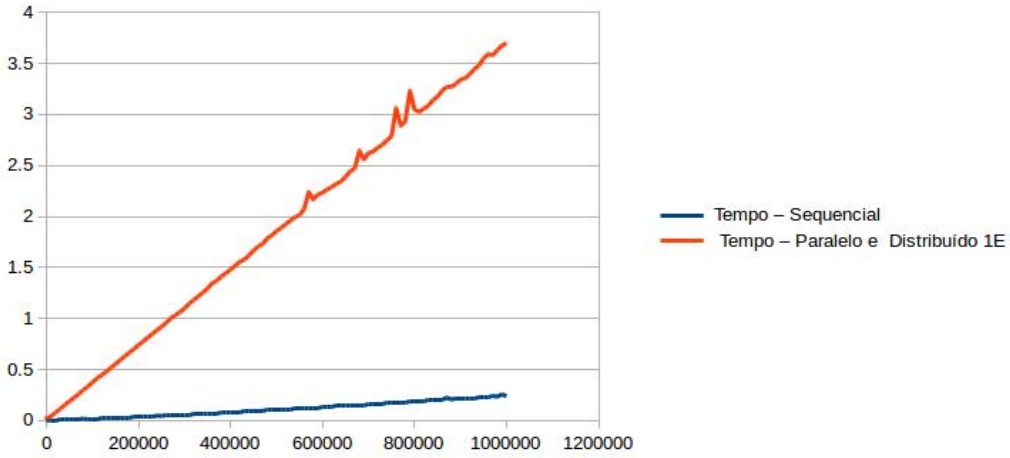

Figura 1: Tempo de execução para 1 Escravo

Observando a figura 2 e 3 não vemos uma diminuição significativa do tempo de reposta, isso pode ser dado pelo fato que a computação de ordenação não demora tanto para acontecer.

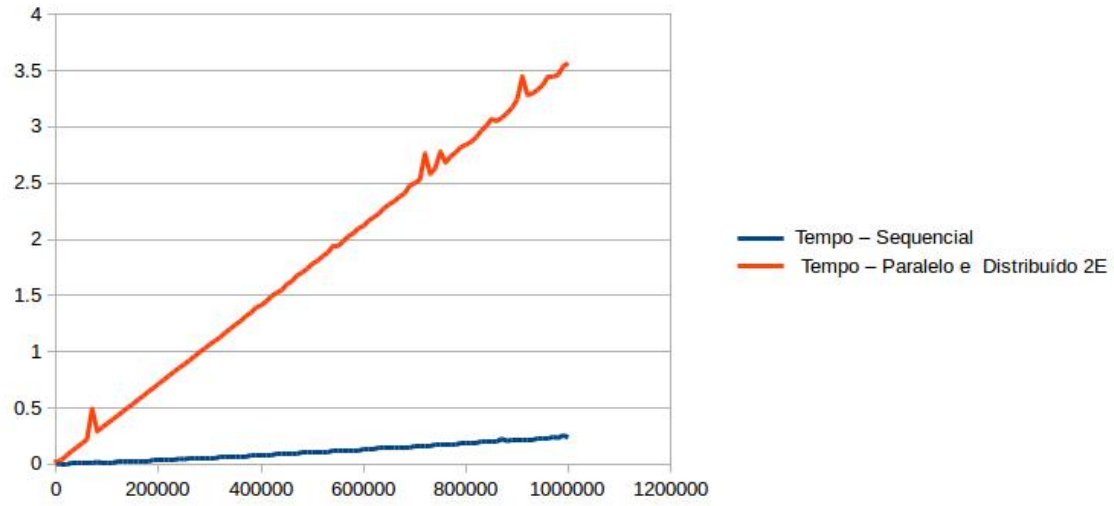

Figura 2: Tempo de execução para 2 Escravos

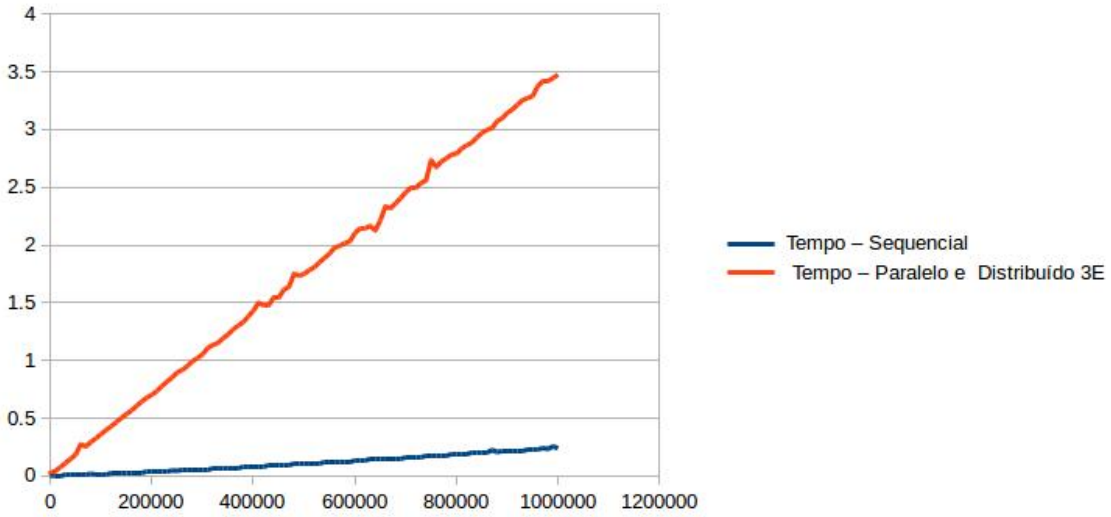

Figura 3: Tempo de execução para 3 Escravos

Pela figura 4 percebemos a semelhança da solução com os casos anteriores, pois o tempo total é bem semelhante para todos os casos.

A pouca diferença do tempo pode ser explicada observando o tempo do caso sequencial que é pequeno, e nesse caso a paralelização iria diminuir esse tempo já pequeno. A demora é dada pela comunicação do middleware e pela comunicação entre os computadores usando a rede.

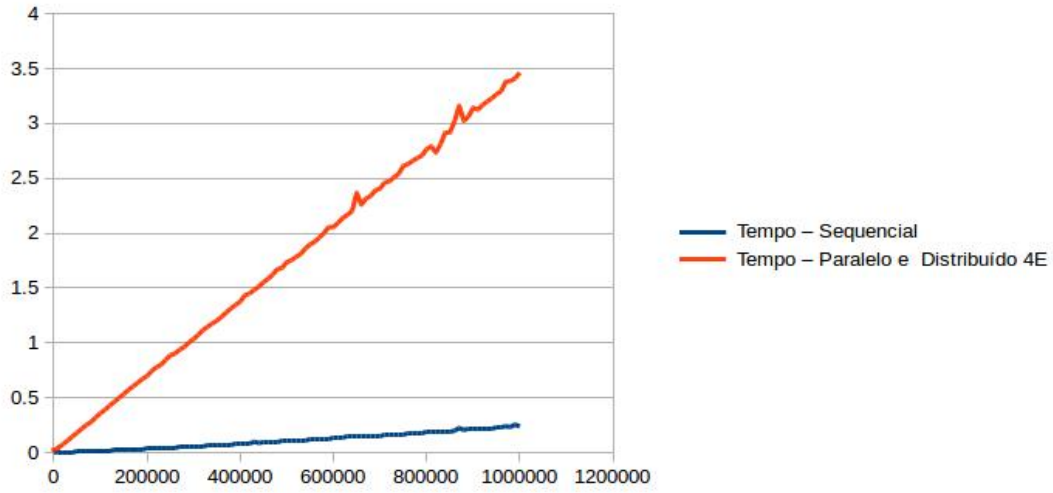

Figura 4: Tempo de execução para 4 Escravos

Mesmo com pouca diferença de tempo podemos ver que a cada escravo adicionado ao tempo de resposta diminui.

3.2. Solução Paralela na Mesma Máquina

A figura 5 apresenta o resultado obtido ao executar o sistema, composto por: Cliente, Mestre e Escravos, em uma mesma máquina multicore. A configuração da máquina é a mesma descrita no início da seção.

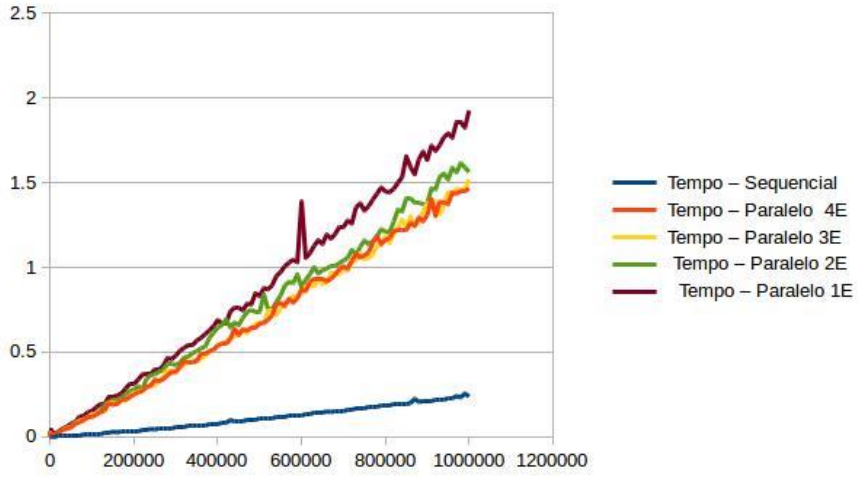

Figura 5: Tempo de execução para a solução paralela na mesma máquina

Observamos que a solução paralela com 4 escravos foi a mais rápida pelo seu maior número de escravos, mas mesmo assim o tempo ainda é menos que o tempo da solução sequencial. Essa diferença de tempo pode ser explicado pelo do uso de um middleware para fazer a comunicação do cliente com o mestre e do mestre com o escravo. Com esse tipo de solução, para resolver o problema de ordenação, haverá várias *threads* rodando no mesma máquina, onde cada *thread* representa um escravo diferente rodando em paralelo com os outros. Logo, com a divisão do trabalho por várias threads diferentes no mesmo computador, tem-se uma diminuição no tempo de resposta ao adicionar mais escravos.

3.3. Análise

Pela figura 6 podemos ver uma comparação das soluções distribuídas. Como falado anteriormente não há uma diferença significativa do tempo de computação, mas podemos observar uma pequena diminuição do tempo a cada escravo adicionado.

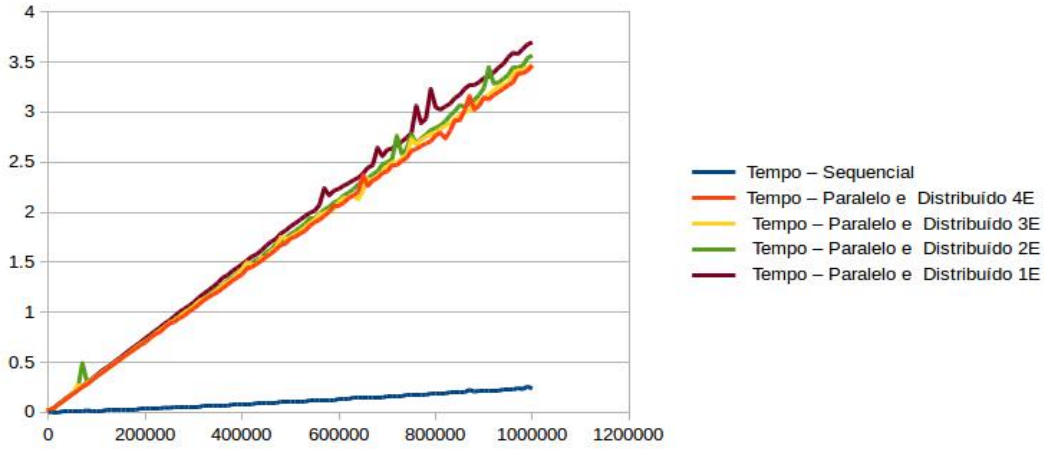

Figura 6: Comparação da solução distribuída

Na seguinte comparação do tempo de resposta da solução distribuída para a local ambas tem um aumento linear no tempo de resposta. Podemos ver a diferença do tempo gerado pelo uso do middleware, na diferença da solução sequencial para a paralela. E também podemos ver o impacto da rede se compararmos a solução paralela com a distribuída.

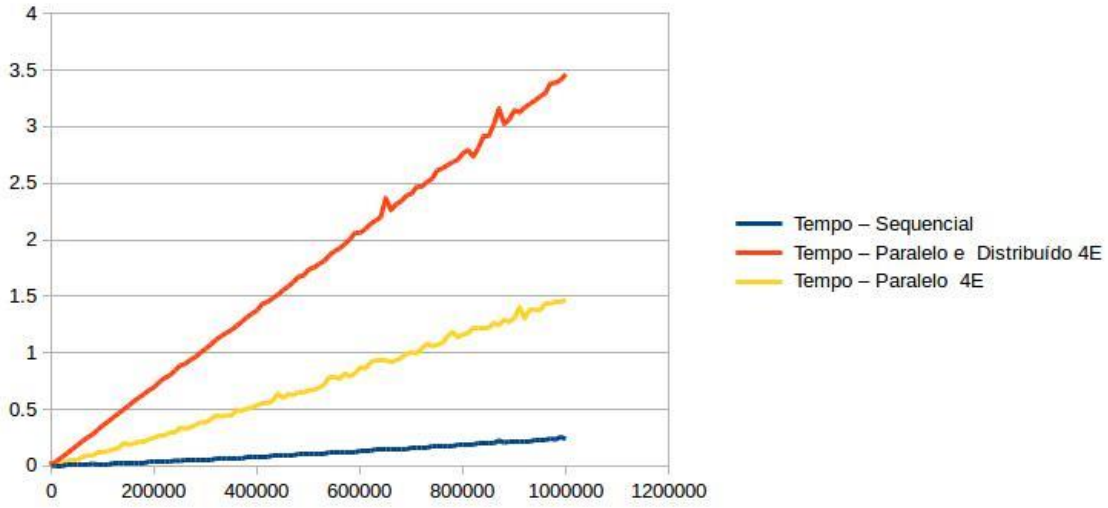

Figura 7: Comparação da solução distribuída com a distribuível

Podemos observar a influência da rede na figura 8, onde a diferença do tempo da solução distribuída para a paralela informa quanto a solução demora a mais para executar levando em conta a rede. Vale a pena informar que no dia em que o experimento foi rodado o LabGrad estava com quase todas as suas máquinas sendo utilizadas no momento.

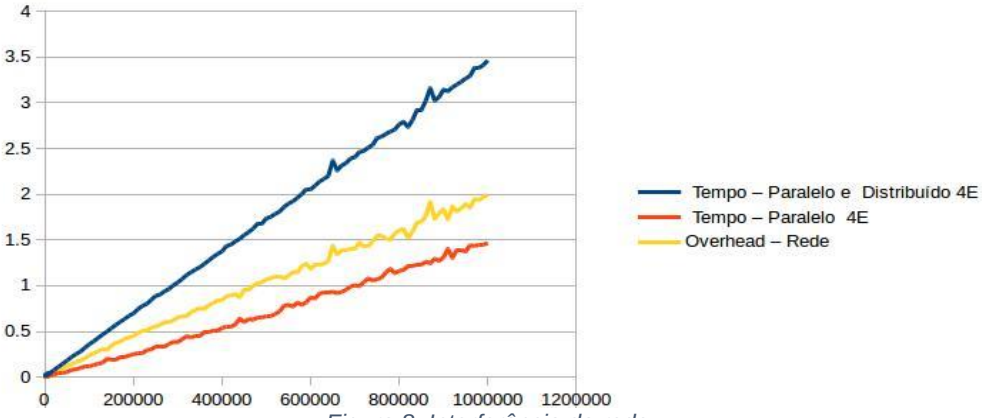

Figura 8: Interferência da rede

3.4 Speed-up

O speed-up nesse trabalho é considerado o tempo de execução da solução sequencial sobre o tempo de execução da solução distribuída. Senso maior que 1 quer dizer que a solução distribuída é executada em tempo menor,

caso contrário ela é executada em tempo menor que a solução sequencial. A figura 9 apresenta o speed-up calculado.

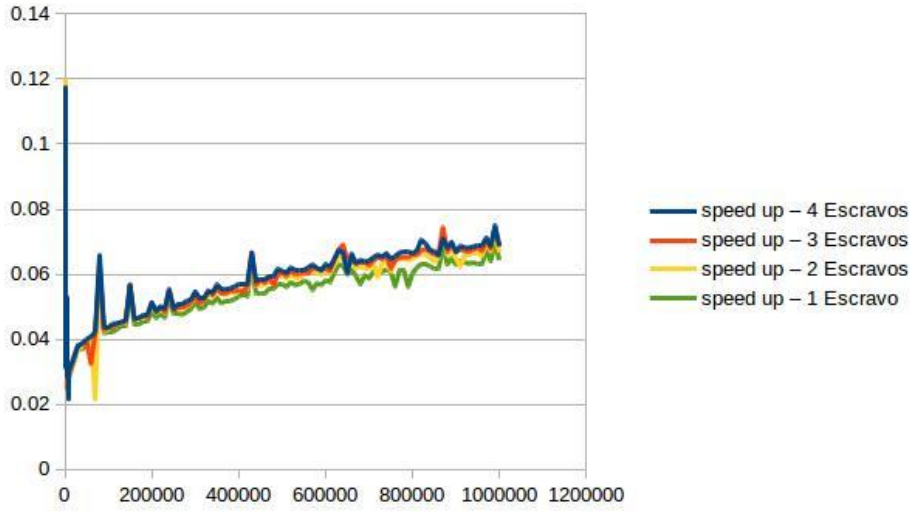

Figura 9: speed up

Observamos que o fator de speed-up sempre se manteve menor que 1, a solução distribuída sempre foi mais lenta. Isso foi causado pelo overhead de comunicação, da rede e do middleware. Apesar do resultado podemos concluir que com um aumento de escravos temos uma melhora do speed-up.

4. Overhead de Comunicação

Durante a execução de computação de forma paralela e distribuída, um dos gargalos que se enfrenta é a de transmissão de dados. Tempo de overhead é o tempo gasto durante toda e qualquer transmissão de dados durante a execução do processo, ou seja, é todo o tempo onde a computação realizada não é considerada como “útil” para a resolução do problema. Além disso, também está incluso nesse tempo a criação de *threads*, a espera por essas threads e etc.

Uma forma de exemplificar isso é o momento onde o cliente efetua uma requisição de ordenar um determinado vetor para o mestre. Ao executar tal tarefa, o cliente precisa enviar para o mestre o vetor para ordenar e esse overhead entra na história. Ao enviar o vetor para o mestre, o cliente tem que submeter esses dados via rede para o mestre, que por sua vez precisa enviar

subvetores do vetor original para cada escravo. Os escravos por sua vez precisam devolver o subvetor ordenado ao mestre que tem que devolver o vetor (agora completamente ordenado) ao cliente. Logo, do tempo total gasto para ordenar o vetor utilizando a arquitetura mestre/escravo de computação paralela e distribuída, grande parte do tempo foi gasto com overhead.

Para poder aproximar o valor do tempo de overhead, foram implementados métodos no mestre e nos escravos que desconsideram a “computação útil” e se preocupam apenas com o tempo de overhead. Por exemplo, quando uma requisição chega ao mestre, ele divide o vetor e repassa aos escravos que por sua vez não fazem nada a não ser apenas retornar para o mestre o próprio vetor que receberam. Já o mestre, não efetua o *merge* entre cada parte de cada escravo, ele apenas devolve ao cliente o vetor originalmente solicitado para ser ordenado. Com isso, tem-se aproximadamente todo o tempo gasto apenas com transmissão de dados.

Abaixo, seguem alguns gráficos exemplificando esses dados levando em consideração o tamanho do vetor a ser ordenado e a quantidade de escravos variando de 1 até 5.

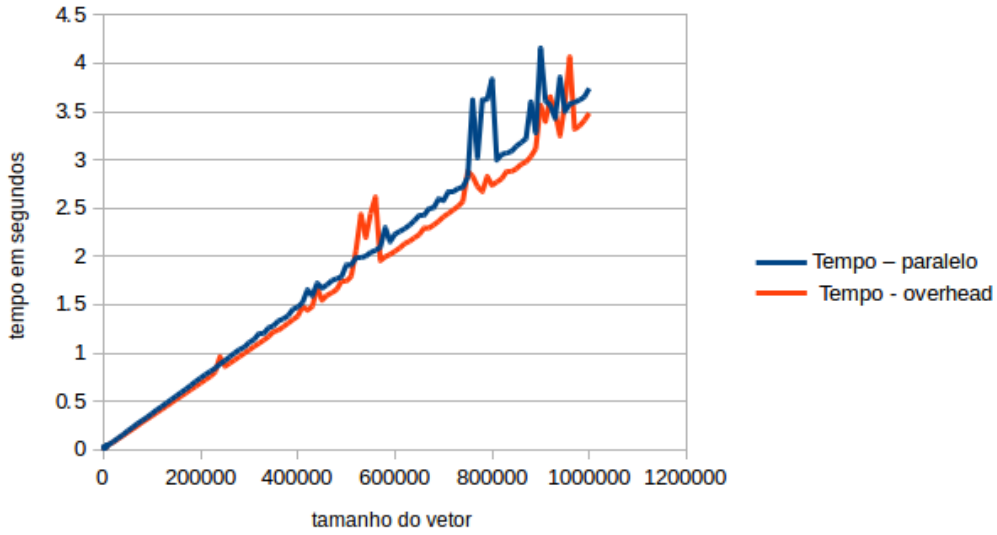

Figura 10: Overhead – 1 Escravo

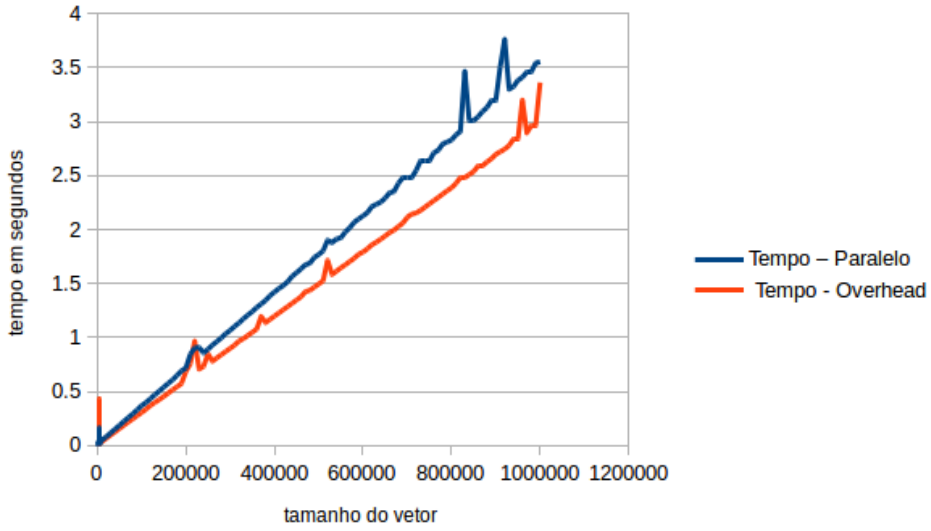

Figura 11: Overhead - 2 Escravos

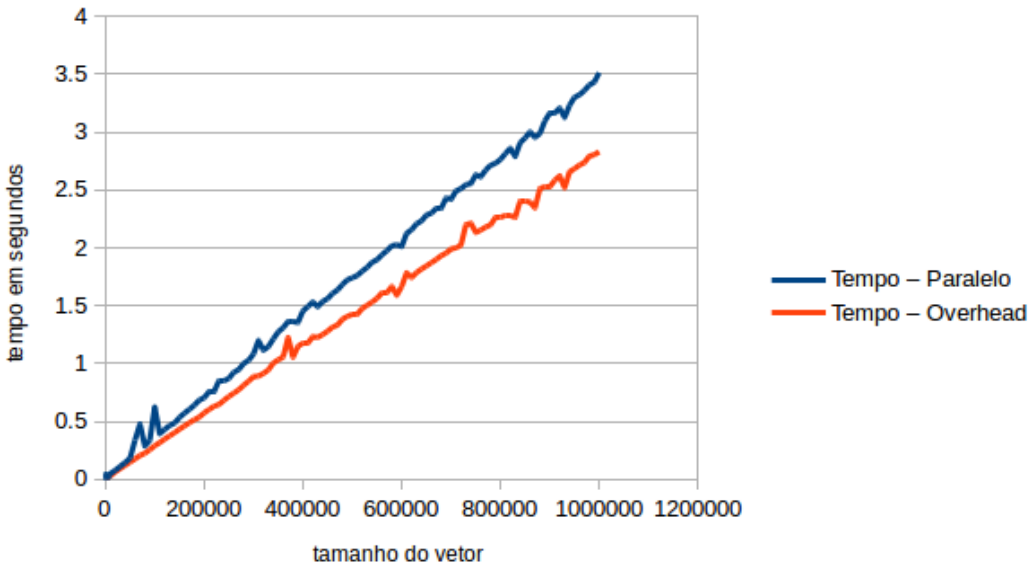

Figura 102: Overhead - 3 Escravos

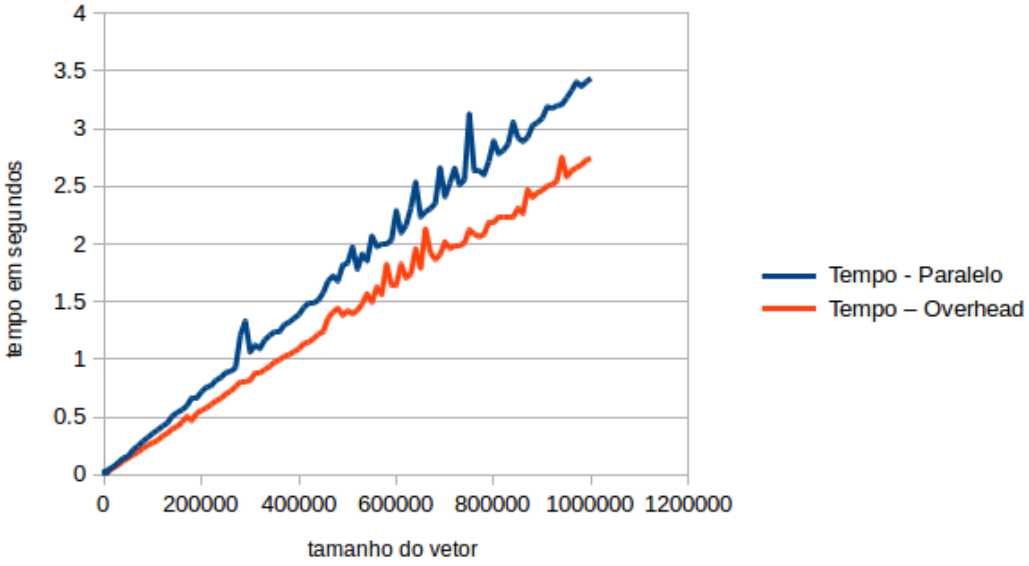

Figura 13: Overhead - 4 Escravos

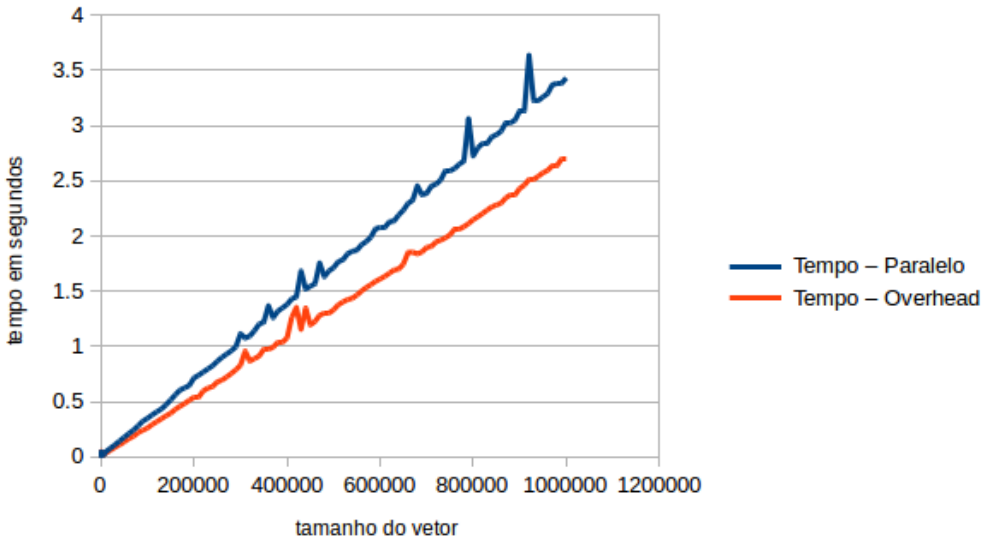

Figura 11: Overhead - 5 Escravos

Analisando os dados apresentados nos gráficos (Figura10, Figura 11, Figura 12, Figura 13 e Figura 14), fica claro o tamanho do impacto do overhead no tempo total da computação. Quanto menor é quantidade de elementos a serem ordenados, maior é a porcentagem de tempo gasto por overhead, mas mesmo em um caso que se tem 1000000 elementos no vetor, fica evidente que o overhead é responsável por grande parte do tempo do processo como um todo. Uma outra coisa que pode ser observado nos gráficos, são os picos de tempo em determinados casos. Esses picos são irrelevantes para um estudo mais completo, uma vez que representam momentos de sobrecarga da rede ou alguma inconsistência do tipo.

Segue uma tabela da média de porcentagem de tempo gasta em overhead utilizando 1, 2, 3, 4 e 5 escravos considerando diferentes tamanhos de vetores como mostrados nos gráficos acima.

	Quantidade de escravos				
	1	2	3	4	5
Porcentagem de tempo gasto com overhead	94,43%	84,53%	81,24%	78,88%	77,86%

Tabela 1: Média da porcentagem de tempo gasto com overhead efetuando 200 testes com tamanho de vetores variando de 1 até 1000000

Ao analisar as porcentagens máximas de tempo perdido em overhead fica claro que ao utilizar cinco escravos ao invés de um, o tempo de computação “não útil” diminui, isso se deve ao fato de o mestre dividir o vetor em porções menores, e logo, a rede se torna menos utilizada por cada escravo. Já no caso de um único escravo, o mesmo vetor que chega para o mestre vindo do cliente, não importa o quão grande seja, vai diretamente para o escravo. Logo, pensando no maior tamanho de vetor utilizado durante os experimentos, com 100000 posições, a transferência desses dados é feita quatro vezes pelo mesmo processo. Uma vez do cliente para o mestre, do mestre para o escravo, do escravo para o mestre e do mestre para o cliente. Assim, antes de o escravo começar a computação “útil” ele tem que esperar todo o vetor ser transferido pela rede. No caso da utilização de cinco escravos, diferentes processos seriam responsáveis por transmitir partes menores do vetor por vez, e assim, a partir do momento que o escravo receber sua parte, já pode começar a realizar as operações necessárias para resolver o problema.

5. Conclusão

Ao fim de todos os experimentos e inúmeros testes, foi fácil observar que para o problema proposto, de ordenar vetores, a solução paralela não foi tão bem-sucedida quanto a solução distribuída. O principal fator que atrapalha a solução paralela, é sem dúvida, o tempo de overhead que engloba toda a

transmissão, *threads* e coisas do gênero. Tal tempo, representa uma porção muito grande sobre o tempo total de computação.

Ao adicionar a quantidade de escravos, foi possível verificar que a porcentagem do tempo de overhead foi decaindo, confirmando a nossa hipótese inicial de que como o tamanho do vetor seria menor, o tempo gasto para transmissão para cada escravo seria menor e essa transmissão é executada de forma “paralela”.

Ficou claro, que para a utilização de processamento paralelo e distribuído deve ser realizado um estudo prévio para cada caso analisando o total de tempo perdido com overhead, que pode aumentar e muito o tempo de processamento do mesmo dado comparado com a forma essencial.

Percebemos que para esse problema a solução serial seria melhor analisando o fator de speed-up, em comparação com a solução paralela. Onde o fator foi sempre menor que e isso 1 mostrou que o overhead teve um peso maior que qualquer ganho de processamento que esse problema teve. O que podemos confirmar, como estudado, é que para um problema de paralelização caso se perca mais tempo com a comunicação de dados, uma possível paralelização pode não apresentar muita vantagem.