

Alexandre Yuji Kajihara

Similaridade entre tweets usando Open MPI

Relatório técnico de atividade prática solicitado pelo professor Rodrigo Campiolo na disciplina de Sistemas Distribuídos do Bacharelado em Ciência da Computação da Universidade Tecnológica Federal do Paraná.

Universidade Tecnológica Federal do Paraná – UTFPR

Departamento Acadêmico de Computação – DACOM

Bacharelado em Ciência da Computação – BCC

Campo Mourão

Dezembro / 2017

Resumo

O Open MPI (*Message Passage Interface*) é um padrão desenvolvido para fornecer uma API (*Application Programming Interface*) para um conjunto de operações de passagem de mensagens. O nosso projeto é verificar a similaridade entre tweets de uma base de dados, utilizando o Open MPI, que utiliza o conceito de mestre-escravo em que o mestre irá designar para os escravos verificar quais tweets são similares. De início foi feita a arquitetura que utiliza o conceito de mestre-escravo, logo após, foi implementado uma aplicação utilizando a API do Twitter em Python, para que se pudesse escolher tweets com uma palavra desejada pelo usuário para verificar a similaridade entre os tweets desse resultado. Após isso, com várias bases de dados começamos a implementação em C na qual lemos o arquivo no formato JSON (JavaScript Object Notation) com os tweets, desenvolvemos o índice de Jaccard, que será utilizado para verificar a similaridade entre eles e por fim, implementamos a parte da comunicação do mestre-escravo, para que o mestre ordene para cada escravo qual parte do arquivo deve ser analisado. O resultado é que conseguimos executar a nossa aplicação tanto localmente e com um outro computador em uma rede local, desempenhando o papel de escravo. Concluímos que conseguimos notar bem a diferença quando executamos localmente e quando temos um mestre e um escravo, em que nesse último caso o uso da CPU (*Central Process Unit*) no mestre fica inferior ao do escravo que tem um uso muito mais elevado, já que o escravo está realizando as operações.

Palavras-chave: rede de computadores. LAN (*Local Area Network*). SSH (*Secure Shell*). NFS (*Network File System*). jQuery.

Sumário

1	Introdução	4
2	Objetivos	4
3	Fundamentação	4
4	Materiais	6
5	Procedimentos e Resultados	7
6	Discussão dos Resultados	11
7	Conclusões	11
8	Referências	11

1 Introdução

Para realizar esse projeto foi necessário compreender como iríamos verificar a similaridade entre tweets, como iríamos utilizar a ferramenta distribuída e como seria definido a arquitetura. Então assimilamos que teríamos uma base de dados com inúmeros tweets, em que iríamos verificar quais desses são similares, que teríamos uma ferramenta que iria permitir a comunicação entre processos que no nosso caso é o Open MPI (*Message Passage Interface*) em que temos o conceito de mestre-escravo, ou seja, o mestre manda o escravo realizar alguma tarefa. Sabendo a ideia do Open MPI, nos auxiliou a realizar a nossa arquitetura. Além disso, foi indispensável ter conhecimento de programação, para escrever os algoritmos que tentam verificar a similaridade entre tweets, as funções que devem ser utilizadas para a comunicação do mestre com os escravos. Enfim, também foi necessário saber quais comandos, argumentos deveriam ser utilizados para compilar, executar tanto localmente quanto remotamente, quanto nos outros computadores.

2 Objetivos

O objetivo desse projeto é desenvolver uma aplicação utilizando ferramenta distribuída. No nosso caso a aplicação seria verificar a similaridade entre tweets, a partir de uma base de dados e a ferramenta distribuída, seria o Open MPI (*Message Passage Interface*), que utiliza o conceito de mestre-escravo, em que o mestre manda o escravo verificar a similaridade entre determinados tweets.

3 Fundamentação

Os fundamentos necessários foram saber descrever a arquitetura do nosso projeto, compreender como utilizar a API (*Application Programming Interface*) do Twitter utilizando a linguagem de programação Python, as rotinas do Open MPI (*Message Passage Interface*), como compilar, executar localmente e em outras máquinas e por fim, saber a linguagem de programação C.

Primeiramente instalamos o Python e a API do Twitter para o Python, que no nosso caso foi o tweepy. Os comandos para instalar ambos pode ser visualizado abaixo:

```
$ sudo apt-get install python
$ sudo pip install tweepy
```

Após isso, instalamos o jQuery, pois como a entrada da nossa aplicação é um arquivo no formato JSON, e queremos somente um campo desse arquivo, precisamos instalar pelo seguinte comando:

```
$ sudo apt-get install jq
```

Depois disso, fomos instalar o Open MPI, em que primeiramente fizemos o *download* dele. Com ele no nosso computador acessamos o diretório aonde foi feito o *download* pelo terminal, e executamos os seguintes comandos:

```
$ tar -xzf mpich2-1.4.tar.gz
$ cd mpich2-1.4
$ ./configure --disable-fortran
$ make; sudo make install
```

Nos comandos acima, extraímos o arquivo, abrimos a pasta que foi extraída, configuramos e por fim instalamos. Esse processo foi feito em todas as nossas máquinas e pode demorar alguns minutos dependendo da máquina. Para compilar é muito similar ao GCC (*GNU Compiler Collection*), em que os comandos do Open MPI que devem ser utilizados são:

```
$ mpicc main.c -o main
```

O mpicc é o compilador, o segundo é o arquivo escrito na linguagem de programação C, o “-o” seria a saída do arquivo e o nome do arquivo de saída. Já para executar localmente, o comando é o que está abaixo:

```
$ mpirun -np X ./main
```

O mpirun é utilizado para executar, o “-np” seria para especificar a quantidade de processos, “X” seria a quantidade de processos e por fim, seria o nome que foi dado para o arquivo de saída na etapa anterior. Agora para executar em outro computador são vários processos. Primeiramente, precisamos executar os comandos abaixo em todas as máquinas desejadas:

```
$ sudo apt-get install openssh-server
$ ssh-keygen -t dsa
$ ssh-copy-id Y
$ eval $(ssh-agent)
$ ssh-add ~/.ssh/id_dsa
$ ssh Y
```

Primeiramente é necessário instalar o SSH (*Secure Shell*), depois geramos uma chave e copiamos para um certo endereço IP (*Internet Protocol*), e pra finalizar adicionamos ele. O “Y” é o endereço IP, ou seja, se você é o computador um você deve adicionar o IP do computador dois, e se você for computador dois você deve adicionar o IP do seu computador um. Após isso, devemos definir quem é o mestre e quem é o escravo. Do lado do mestre, deve ser utilizar os seguintes comandos:

```
$ sudo apt-get install nfs-kernel-server
```

```
$ mkdir Z
$ sudo nano /etc/exports
$ exportfs -a
$ sudo /etc/init.d/nfs-kernel-server start
```

No mestre, primeiramente instalamos o NFS (*Network File System*), depois criamos um diretório com o nome desejado, depois abrimos o arquivo que se encontra em `/etc/exports` e adicionamos “*PATH/Z*(rw, sync, no_root_squash, no_subtree_check)*”, onde o *PATH/Z* deve ser substituído pelo local onde a pasta foi criada na etapa anterior. O penúltimo, comando está relacionado toda vez que há mudança nesse diretório e por fim inicializamos. Já do lado do escravo, deve ser utilizar os seguintes comandos:

```
$ sudo apt-get install nfs-common
$ mkdir Z
$ sudo mount W:Z PATH/Z
```

No escravo, primeiramente instalamos o NFS, criamos um diretório com o mesmo nome que está no mestre e por fim montamos, em que *W* é o endereço IP (*Internet Protocol*) do mestre o *Z* é a localização do diretório no mestre e *PATH/Z* a localização do diretório no escravo. Para compilar e o mesmo processo a diferença fica na execução, que é executada pelo comando abaixo:

```
$ mpirun -np A -hosts B ./main
```

O “*A*” deve ser substituído pela a quantidade de processos e o “*B*” deve ser substituído pelo endereço do escravo. Caso não saiba o endereço IP pode ser obtido pelo primeiro comando abaixo, ou se não está presente pode ser instalado pelo segundo comando.

```
$ mpirun -np A -hosts B ./main
$ sudo apt-get install net-tools
```

4 Materiais

Os materiais necessários foram dois *laptops* um deles com as seguintes configurações:

- Sistema Operacional Ubuntu 17.04 64 bits
- Intel Core i7-6500U Quad-Core - 2.50Ghz
- 16 GB de memória RAM (*Random Access Memory*)
- Placa de vídeo NVIDIA Geforce 940M com 4GB

- OpenMPI (*Message Passage Interface*), compilador para linguagem de programação C na versão 1.4, que permite utilizar as rotinas do Open MPI
- Python compilador para linguagem de programação Python na versão 2.7.13
- Editor de texto Sublime Text 3143

E o outro *laptop* com a seguinte configuração:

- Sistema Operacional Ubuntu 17.04 64 bits
- Intel Pentium(R) CPU 2117U Dual-Core - 1.80GhZ
- 4 GB de memória RAM (*Random Access Memory*)
- Placa de vídeo Intel Ivybridge Mobile
- OpenMPI (*Message Passage Interface*), compilador para linguagem de programação C na versão 1.4, que permite utilizar as rotinas do Open MPI
- Python compilador para linguagem de programação Python na versão 2.7.13
- Editor de texto Sublime Text 3143

5 Procedimentos e Resultados

Todas as implementações que iremos descrever à seguir, estão presente em <https://github.com/xaaaandao/Tweets-OpenMPI>. Como a nossa ferramenta distribuída utiliza o conceito de mestre-escravo, montamos a arquitetura que pode ser visualizada na figura 1, em que temos apenas um mestre que gerencia quais tweets cada escravo irá verificar a similaridade. Além do que, as setas do mestre para o escravo representa a mensagem enviada para o escravo de quais intervalos deve ser aplicado o índice de Jaccard e as setas dos escravos para o mestre seriam os tweets que são similares.

Após definida a arquitetura implementamos um programa que conseguisse nos retornar uma lista de tweets com aquela determinada palavra escolhida pelo usuário. Para tal façanha, utilizamos a API (*Application Programming Interface*) do Twitter na linguagem de programação Python, que dado uma palavra ele retorna tweets com aquela palavra, e nos gera um arquivo de saída no formato JSON. Essa aplicação foi integrado a nossa aplicação em linguagem C.

Com algumas entradas que nos foram fornecidas no formato JSON, começamos a realizar o programa na linguagem C. A primeira etapa, foi ler o arquivo de entrada, porém antes de realizar a leitura tivemos que utilizar o comando “jq” no arquivo de entrada, pois como estávamos interessados somente no campo que continham o texto do tweet. Essa

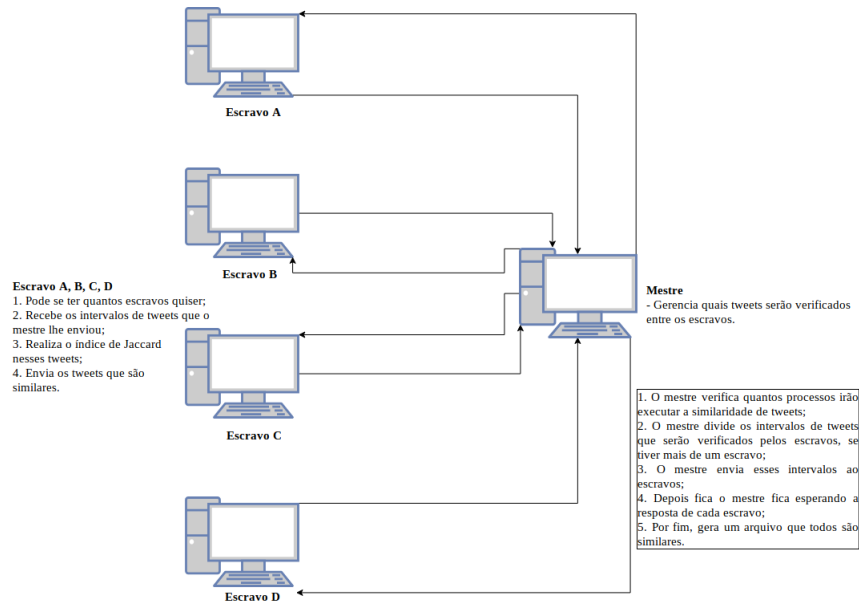


Figura 1: Arquitetura do nosso projeto

etapa gerava um arquivo, e com esse arquivo nosso programa em C lia todas as linhas e armazenava em uma lista. A cada tweet inserido na lista foi dado um identificador, foi feito cópia do tweet original, foi feita outra cópia do tweet original só que removemos palavras e caracteres indesejados, e também armazenamos a quantidade de palavras do tweet original e do tweet após a remoção, todos esses campos presentes em cada um dos nós da lista, irão nos auxiliar nas etapas superiores.

Com todos os tweets carregados na lista, percorremos ela e para cada nó dela, comparamos com os outros demais nós da lista e para cada comparação aplicamos o índice de Jaccard. Essa verificação consiste em fazer a divisão da intersecção, que seria palavras que estão presentes em ambos os tweets e pela união que seria a quantidade de palavras presente em ambos os tweets. Essa divisão apresenta um valor entre zero e um, e para valores que essa divisão é maior que 0.3 adicionamos em uma lista, que contém os tweets que são similares, o identificador de ambos, o valor de intersecção, união e similaridade.

Até essa parte do projeto só estávamos executando localmente e com passando o valor de um para o argumento “-np” para executar o nosso programa. No momento que passamos a utilizar valores mais alto para o argumento, a diferença foi notória, em que o tempo de execução caiu menos da metade, como pode ser visto na figura 2 e 3.

Após executar localmente, fomos implementar o Open MPI (*Message Passage Interface*). O primeiro processo feito nessa etapa, foi diferenciar quem é o mestre e quem o escravo, isso feito é a partir de um condicional em que se for zero o valor do *rank* é mestre, caso contrário é escravo.


```
xandao@tchatcha: ~/cloud/Tweets-OpenMPI
Tweets na lista
Aplicando Índice de Jaccard (t1:499-t2:498/t:500)
Contando as intersecções dos dois tweets
Contando as uniões dos dois tweets
Arquivo de saída gerado com sucesso! (arquivo: output.txt)

real    4m16,975s
user    0m6,744s
sys     0m36,420s
xandao@tchatcha:~/cloud/Tweets-OpenMPI$
```

Figura 2: Execução utilizando o valor de “-np” sendo um, utilizando a função time do Ubuntu e uma base de dados de 500 tweets

```
xandao@tchatcha: ~/cloud/Tweets-OpenMPI
Tweets na lista
Aplicando Índice de Jaccard (t1:499-t2:498/t:74)
Contando as intersecções dos dois tweets
Contando as uniões dos dois tweets
Arquivo de saída gerado com sucesso! (arquivo: output.txt)

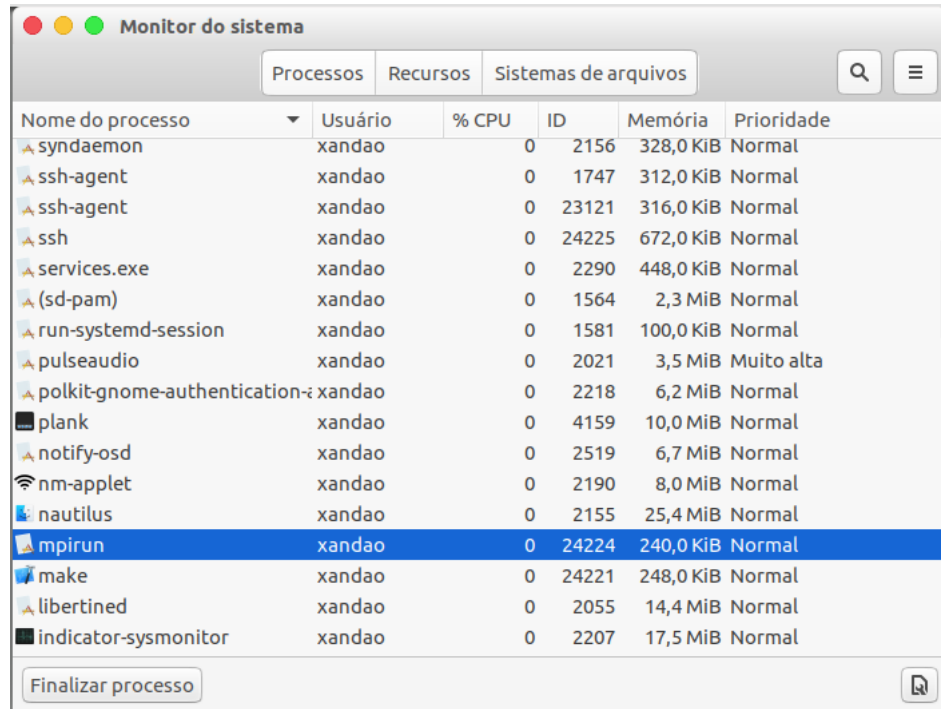
real    0m22,464s
user    0m26,164s
sys     0m7,468s
xandao@tchatcha:~/cloud/Tweets-OpenMPI$
```

Figura 3: Execução utilizando o valor de “-np” sendo oito, utilizando a função time do Ubuntu e uma base de dados de 500 tweets.

Então definido quem era o mestre e o escravo, desenvolvemos primeiro a parte do mestre. No mestre verificamos a quantidade de processos desejado, se fosse um só significa que ele mesmo vai ter que executar, se for dois processos, significa que teremos um mestre e um escravo então só o único escravo executará, agora quando o valor é superior a dois, eu vou dividindo em partes iguais a quantidade total de tweets e mando o intervalo de tweets para cada escravo verificar a similaridade. Após enviar todos os intervalos, o mestre fica aguardando a resposta em uma *thread*.

Cada escravo recebe a parte que lhe foi encaminhada, e aplica o índice de Jaccard

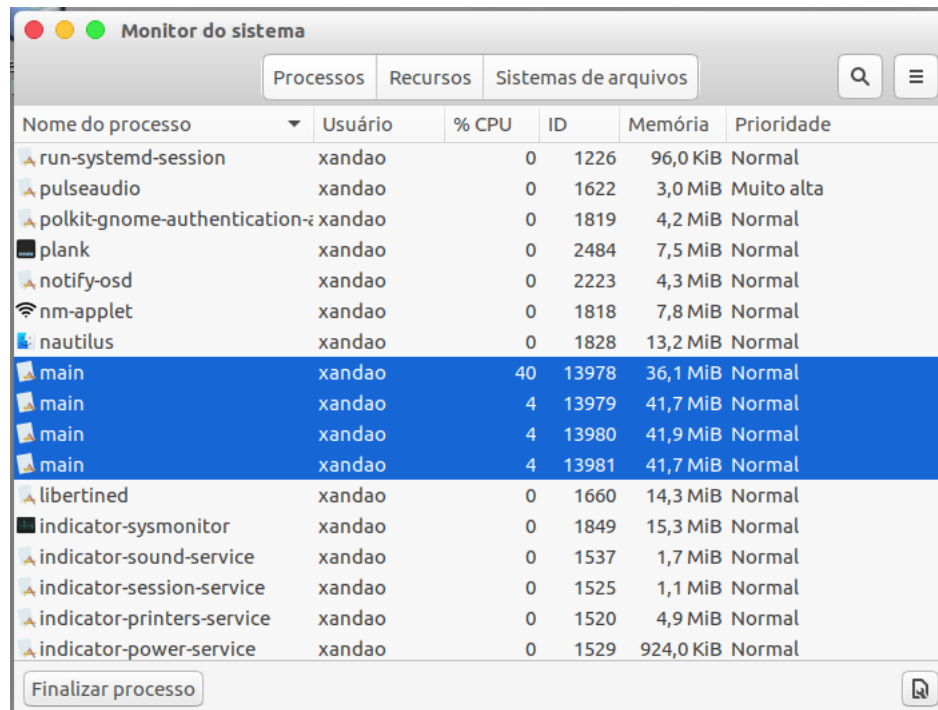
naquele intervalo que lhe foi enviado. Depois de aplicar, enviamos ao mestre os identificadores de todos os tweets que são similares. Podemos ver a execução no mestre na figura 4 e a execução no escravo na figura 5.



Nome do processo	Usuário	% CPU	ID	Memória	Prioridade
syndaemon	xandao	0	2156	328,0 KiB	Normal
ssh-agent	xandao	0	1747	312,0 KiB	Normal
ssh-agent	xandao	0	23121	316,0 KiB	Normal
ssh	xandao	0	24225	672,0 KiB	Normal
services.exe	xandao	0	2290	448,0 KiB	Normal
(sd-pam)	xandao	0	1564	2,3 MiB	Normal
run-systemd-session	xandao	0	1581	100,0 KiB	Normal
pulseaudio	xandao	0	2021	3,5 MiB	Muito alta
polkit-gnome-authentication-	xandao	0	2218	6,2 MiB	Normal
plank	xandao	0	4159	10,0 MiB	Normal
notify-osd	xandao	0	2519	6,7 MiB	Normal
nm-applet	xandao	0	2190	8,0 MiB	Normal
nautilus	xandao	0	2155	25,4 MiB	Normal
mpirun	xandao	0	24224	240,0 KiB	Normal
make	xandao	0	24221	248,0 KiB	Normal
libertined	xandao	0	2055	14,4 MiB	Normal
indicator-sysmonitor	xandao	0	2207	17,5 MiB	Normal

Finalizar processo

Figura 4: Execução no lado do mestre.



Nome do processo	Usuário	% CPU	ID	Memória	Prioridade
run-systemd-session	xandao	0	1226	96,0 KiB	Normal
pulseaudio	xandao	0	1622	3,0 MiB	Muito alta
polkit-gnome-authentication-	xandao	0	1819	4,2 MiB	Normal
plank	xandao	0	2484	7,5 MiB	Normal
notify-osd	xandao	0	2223	4,3 MiB	Normal
nm-applet	xandao	0	1818	7,8 MiB	Normal
nautilus	xandao	0	1828	13,2 MiB	Normal
main	xandao	40	13978	36,1 MiB	Normal
main	xandao	4	13979	41,7 MiB	Normal
main	xandao	4	13980	41,9 MiB	Normal
main	xandao	4	13981	41,7 MiB	Normal
libertined	xandao	0	1660	14,3 MiB	Normal
indicator-sysmonitor	xandao	0	1849	15,3 MiB	Normal
indicator-sound-service	xandao	0	1537	1,7 MiB	Normal
indicator-session-service	xandao	0	1525	1,1 MiB	Normal
indicator-printers-service	xandao	0	1520	4,9 MiB	Normal
indicator-power-service	xandao	0	1529	924,0 KiB	Normal

Finalizar processo

Figura 5: Execução no lado do escravo.

O resultado que obtivemos foi que encontramos os tweets que são similares, dado

uma base de dados que são pequenas. Além disso, conseguimos executar tanto localmente quanto em outro computador dentro de uma rede local.

6 Discussão dos Resultados

Analisando a nossa aplicação de similaridade, percebemos que conseguimos encontrar uma grande quantidade de tweets que são similares. Porém, durante a realização desse projeto notamos que quanto mais removemos caracteres e palavras indesejadas dos tweets originais, mais tweets seriam similares.

Um fato que notamos quando designávamos para o escravo realizar a similaridade, é que a carga da CPU (*Central Processing Unit*) no escravo fica elevado, enquanto do mestre não ficava tanto elevado. Acreditamos que isso poderia ser melhorado se utilizássemos mais escravos para tentar encontrar a similaridade entre os processos, ou se melhorasse o nosso código, pois existem funções que executam em n^2 , e com algoritmos mais eficiente e dividindo a carga conseguiríamos processar bases de dados maiores em um tempo menor.

7 Conclusões

Acreditamos que conseguimos visualizar muito bem o papel do Open MPI (*Message Passage Interface*), que seria as trocas de mensagens, entre o mestre e o escravos, quando o mestre manda mensagens com os intervalos que devem ser verificados as similaridades, e o escravo recebe a mensagem e devolve o identificadores dos tweets que são similares. Enfim, identificamos na prática o MPI que é um padrão para comunicação de dados em computação paralela, na qual existem várias máquinas rodando (que no nosso caso é um mestre e um escravo), e tudo isso aparenta ser apenas um máquina rodando.

8 Referências

COLOURIS JEAN DOLLIMORE, T. K. G. B. G. Sistemas Distribuídos: Conceitos e Projeto. 5. ed. [S.l.]: Bookman, 2013. ISBN 978-85-8260-054-2.