Implementação do algoritmo Map/Reduce

Trabalho prático de avaliação No. 2

Universidade de Aveiro
Licenciatura em Engenharia Informática
UC 40382 - Computação Distribuída
Junho de 2019 - 2º Semestre

Trabalho realizado por:

88808 - João Miguel Nunes de Medeiros e Vasconcelos
88886 - Tiago Carvalho Mendes

Código do projeto disponível em:

https://github.com/detiuaveiro/map-reduce-cd_p4_88808_88886

universidade de aveiro deti departamento de electrónica, telecomunicações e informática

1. Introdução

O presente documento tem como principal objetivo descrever os pontos principais da solução desenvolvida pelos alunos apresentados na capa do mesmo, tendo em conta o problema proposto como segundo trabalho prático de avaliação da unidade curricular de Computação Distribuída da Universidade de Aveiro. Este problema consistia na implementação do algoritmo ***Map/Reduce*** através de um sistema distribuído. É de salientar que a solução foi desenvolvida usando a linguagem de programação Python.

No guião do trabalho prático, era pedido aos alunos a implementação de duas *entidades*: um coordenador e um trabalhador (*worker*). O coordenador tem como principal objetivo partir um texto literário em pedaços (*blobs*) e, primeiramente, distribuir esses pedaços pelos diversos *workers* registados, para que efetuem a tarefa de ***Map.*** Após receber algumas listas de tuplos por parte dos *workers,* o coordenador envia uma lista contendo duas listas recebidas para um dos *workers*, para que seja efetuada a tarefa de ***Reduce***. No final, é desejado construir um histograma com o número de ocorrências de palavras distintas presentes no texto. Era ainda pedido a implementação de um mecanismo de **tolerância a falhas**, caso algum dos processos morresse, em especial a implementação de um processo coordenador de *backup.* É ainda de salientar que a comunicação entre processos deveria ser feita através de sockets TCP, usando threads, selectors ou programação assíncrona.

2. Solução base

Relativamente à solução base, era pedido apenas a comunicação *TCP* entre um **coordenador e um worker** e uma correta implementação das tarefas de Map e Reduce. No final desta etapa, já deveríamos ter um histograma correto do número de ocorrências de palavras distintas do texto literário.

A primeira abordagem utilizada para estabelecer a comunicação TCP foi recorrendo a ***threads***. No entanto, este método revelou-se ineficiente pois, sempre que um novo *worker* estabelecesse uma nova ligação com o *coordenador,* este último inicializava uma nova *thread* para lidar com as mensagens recebidas de apenas um *worker*, o que não era escalável para um elevado número de *workers*. Portanto, decidimos investigar sobre o paradigma de programação assíncrona e implementámos um ***Callback Based Server*** que extende o ***asyncio.Protocol.*** Assim, a nossa única preocupação utilizando esta abordagem era redirecionar uma mensagem recebida na classe deste servidor para a classe do coordenador, ficando este último com a responsabilidade de a filtrar.

Relativamente à implementação da tarefa de ***Map*** por parte dos workers, a primeira tarefa realizada foi remover qualquer tipo de caracteres que não fossem letras (com acentuação incluídas), como por exemplo dígitos, espaços em branco, quebras de linha, underscores, etc. De seguida, todas as palavras foram armazenadas numa lista e, uma a uma, foram introduzidas na primeira posição de um tuplo, tendo a segunda posição o número 1, sendo este tuplo depois adicionado a uma nova lista, que é retornada no final do processo de ***mapping.*** Quanto ao processo de ***Reduce,*** são passadas como argumento duas listas retornadas no processo anterior, ordenadas alfabeticamente. Em primeiro, faz-se a contagem do número de ocorrências da mesma palavra em cada uma das listas, individualmente. De seguida, define-se a lista mais pequena como **principal** e a outra como **secundária.** Isto é feito para que se percorra toda a lista **principal** e se utlize o algoritmo de ***binary search*** para encontrar palavras iguais na lista **secundária**, incrementando a contagem. Por fim, acrescenta-se à lista **principal** as palavras **não encontradas** na lista **secundária,** e é usado o algoritmo de **mergesort** para ordenar a lista final, pois tem complexidade temporal **O(nlog(n)).**

2.1 Resolução do problema de tamanho elevado das mensagens enviadas

Um dos problemas com que nos deparámos na troca de mensagens entre o coordenador e os *workers* foi que, a partir de certo ponto, as listas enviadas para efetuar o ***Reduce*** eram demasiado grandes para serem transportadas numa única mensagem pelas sockets TCP. Após a ponderação de diversas alternativas viáveis, decidimos optar pela seguinte: tanto o coordenador ou qualquer um dos workers, sempre que quiserem enviar uma mensagem, codificam o objeto JSON correspondente e enviam tudo o que têm para enviar através da propriedade das sockets ***sendall().*** No entanto, antes de enviar este objeto, é-lhe acrescentado no final um carater especial, muito utilizado na área das redes, de fim de transmissão (***end of transmission),*** representando em UTF-8 por ***0x04.*** Portanto, do lado da entidade que recebe, irá existir uma procura por este caracter por cada pedaço da mensagem que chega. No caso de esse carater ainda não estar presente num pedaço acabado de receber, esse pedaço é adicionado a um buffer e assim sucessivamente, até que esse caracter seja encontrado e seja certo que a mensagem chegou ao fim.

3. Solução distribuída

No seguimento do ponto anterior, para que o coordinator fosse capaz de distribuir tarefas pelos *workers*, era necessário elaborar um algoritmo de distribuição de trabalho**,** enviando aos *workers* ou um *'map_request'* ou um *'reduce_request'*. Portanto, o nosso pensamento foi o seguinte: o coordenador teria uma lista que armazenava as listas retornadas pelo processo de ***Map***. Sempre que essa lista não tivesse 2 elementos, era enviado um *'map_request'*, com o próximo pedaço de texto a processar. Por outro lado, caso essa lista tivesse pelo menos 2 elementos, era imediatamente enviado um *'reduce_request'* com as duas primeiras listas de palavras. Portanto, quer estejam em processamento 1, 4 ou um número indefinido de *workers*, este algoritmo é capaz de distribuir trabalho por todos de uma forma eficiente. É de realçar que o programa é terminado assim que o último pedaço de texto é processado e o coordenador recebe a contagem de todas as palavras presentes no texto literário.

4. Solução distribuída robusta

Para que a solução fosse robusta, era necessário que esta permitisse um número indefinido de *workers*, e que estes recebessem tarefas por parte do coordenador de forma indistinta. Ora, com o algoritmo já explicado no ponto 3, este requisito foi automaticamente satisfeito, visto que independentemente do número de *workers*, o coordenador é capaz de distribuir as tarefas igualmente.

No entanto, era ainda pedido que caso um segundo processo de coordenador se inicia-se, teria automaticamente de perceber que já existia um coordenador em funcionamento e tornar-se backup deste. Para alcançar este objetivo, cada processo coordenador, sempre que se inicia, tenta efetuar um ***bind*** no endereço ('127.0.0.1', 8765). Caso não seja gerada nenhum exceção, este processo sabe que foi o primeiro a ser iniciado e inicia o seu trabalho como coordenador. Caso contrário, o processo conecta-se ao mesmo endereço, anunciando-se como backup do primeiro coordenador através da mensagem ***'register_backup'***.

Ora, com o processo de backup, o coordenador sempre que distribui uma tarefa por um dos workers, envia o estado atual do texto literário, ou seja, quantos pedaços do texto é que ainda faltam processar para que, caso o coordenador morra, o backup posso assumir as suas funções e continuar o processamento sem perda de informação.

5. Tolerância a falhas

No guião do trabalho prático era pedido que, e no seguimento do ponto anterior, a implementação de um mecanismo de tolerância a falhas, caso qualquer um dos processos morresse. Para tal, era preciso lidar com duas situações distintas: a morte do coordenador ou a morte de qualquer *worker*.

Relativamente à morte do coordenador, e como dito anteriormente, existe um processo de *Backup* que recebe mensagens com o estado atual do processamento do texto literário. Quando o coordenador morre, o processo de *Backup* deixa de receber essas mensagens, e por isso apercebe-se do sucedido. Portanto, ele assume-se como o novo coordenador e inicia um objeto da classe *Coordinator* com o estado do último processamento de texto. Ora, com esta operação, o segundo coordenador efetua um ***bind*** na mesma porta em que o primeiro coordenador estava a ouvir, escondendo assim a morte do primeiro coordenador para com os *workers* e permitir a continuação da distribuição do trabalho. É de realçar que esta solução funciona perfeitamente para o caso de apenas estar em execução um *worker*. No caso de estarem em execução mais *workers*, esta técnica funciona mas a contagem final das palavras não é a correta.

Por outro lado, sempre que um *worker* morre, o método ***eof_received*** é chamado na classe ***EchoProtocol(asyncio.BaseProtocol)*** devido à desconexão de um *worker* com o coordenador. A partir desta função, é chamado o método ***handle_hangup*** no coordenador, de forma a lidar com esta desconexão. Essencialmente o que este método faz é remover a conexão terminada do dicionário de conexões do coordenador, e colocar as mensagens enviadas para o falecido *worker* que não tiveram resposta numa fila de mensagens perdidas. Assim, sempre que o coordenador quiser distribuir trabalho, primeiro verifica se existem mensagens nessa fila de mensagens perdidas, e só em caso contrário é que continua o processamento normal, descrito nas páginas anteriores deste documento. É ainda de salientar que esta técnica funciona sempre, para qualquer número de *workers*.

6. Troca de mensagens

Para melhorar a perceção do modo de funcionamento da nossa solução, de seguida apresenta-se um diagrama com todas as trocas de mensagens existentes.

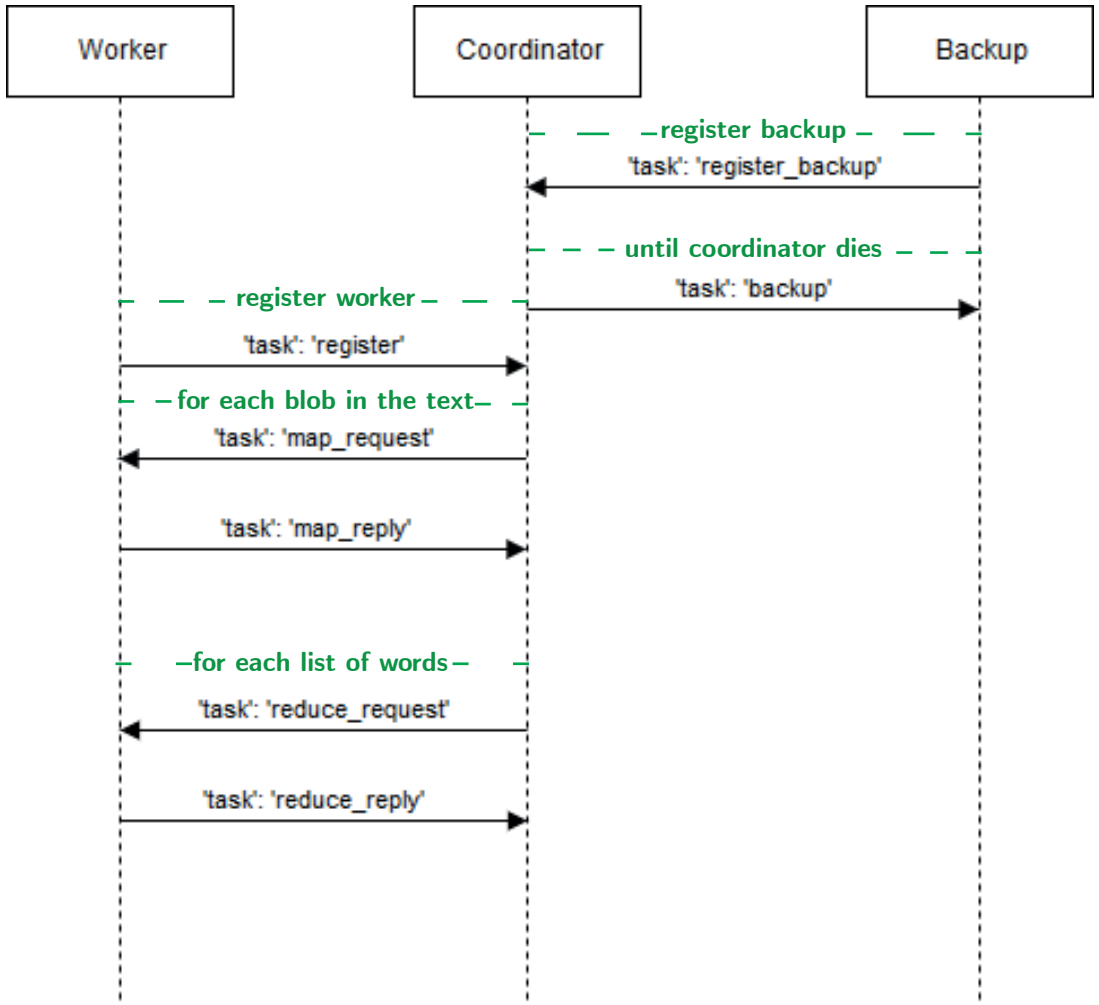

Diagrama 1 - Adaptado do guião do trabalho

7. Análise estatística

Após o desenvolvimento da solução, foi realizada uma análise estatística em várias vertentes. É de realçar que o uso de programação assíncrona para estabelecer as comunicações do coordenador com os workers e dos algoritmos de binary search e merge sort na tarefa de Reduce tornaram a nossa solução mais eficiente do que talvez pelo uso de threads ou de outros algoritmos de ordenação. Na tabela 1 encontram-se representados os tempos de execução da solução para um determinado número de workers e para o texto literário de "Os Lusíadas", desde o ínicio do processo de coordenador até ao fim da contagem. Os valores deram origem ao gráfico 1. Deste gráfico conclui-se que, para apenas um worker, o tempo de execução é muito elevado, tendo um grande decréscimo até ao valor de 4 workers. A partir daqui, os tempos são mais ou menos constantes até a um elevado número de workers (por exemplo, 20), onde se começa a notar uma subida dos tempos de execução. Este facto talvez possa ser explicado pela grande quantidade de mensagens de registo com que o coordenador tem de lidar, tempo esse que poderia ser utilizado para realizar trabalho útil.

No. workers	Tempo (s)
1	23,30
2	8,55
3	4,83
4	4,34
5	4,03
6	3,88
7	4,33
8	4,13
9	4,38
10	4,44
20	7,57
30	9,72

Tabela 1

Gráfico 1

É ainda importante referir que, no processo de recuperação da morte do coordenador e passagem do processo de backup para novo coordenador, os tempos de execução foram semelhantes a estes, apenas com alguns milissegundos de atraso.