Estudo da Plataforma Hadoop Core para Processamento Paralelo via MapReduce

Alexandre Almeida
alexandre.almeida@inf.ufrgs.br

Relatório do Trabalho Final
Programação com Objetos Distribuídos
Programa de Pós-Graduação em Computação
Universidade Federal do Rio Grande do Sul

1 Introdução

O MapReduce é um modelo de programação paralela proposto por [Dean and Ghemawat, 2008], que é integrado a um *framework* responsável por abstrair detalhes da paralelização, tolerância à falhas, distribuição de dados e balanceamento de carga da aplicação. Este modelo de programação/plataforma foi criado com o objetivo de expressar computações simples, porém que devem ser realizadas sobre um grande volume de dados. O *framework* divide a entrada em diversas tuplas $\langle chave, valor \rangle$, sendo estas submetidas à uma função de mapeamento (*map*) que gera um conjunto intermediário de pares $\langle chave, valor \rangle$. Os pares intermediários são submetidos à uma função de redução (*reduce*) responsável por processar todos os valores de uma mesma chave, gerando um novo conjunto final de pares $\langle chave, valor \rangle$.

O Apache Hadoop é um projeto que desenvolve plataformas para programação de aplicações distribuídas, escaláveis, e com tolerância a falhas. Este trabalho tem um enfoque na plataforma Hadoop Core, que suporta o modelo de programação distribuído MapReduce, além de todas as características de controle do paralelismo e de tolerância à falhas descritas por [Dean and Ghemawat, 2008].

O objetivo deste trabalho é avaliar a API de programação do Hadoop Core, bem como fazer um estudo do funcionamento desta plataforma de programação distribuída.

O presente relatório é estruturado como segue. Na Sessão 2 é explicado o modelo de programação MapReduce por meio de dois exemplos; na Sessão 3 a plataforma Hadoop Core é apresentada, fazendo-se um *overview* de sua arquitetura; na Sessão 4 apresenta-se a API básica de programação da plataforma Hadoop Core, bem como os passos para executar uma aplicação; as considerações finais são apresentadas na Sessão 5.

2 Modelo de programação MapReduce

Uma aplicação no modelo MapReduce tem como entrada um conjunto de pares $\langle chave, valor \rangle$, e gera como saída um conjunto de pares final $\langle chave, valor \rangle$. O processamento da aplicação deve ser expresso por uma função de mapeamento e uma função de redução, ambas definidas pelo usuário.

A função de mapeamento recebe o conjunto de pares $\langle chave, valor \rangle$ de entrada, e os mapeia para outro conjunto intermediário de pares compostos por *chave* e *valor*. A chave do conjunto intermediário não precisa ser necessariamente a mesma chave do conjunto de entrada. Por exemplo, é comum que

o *valor* do conjunto de entrada seja usado para gerar as chaves do conjunto intermediário, que serão associadas a um determinado valor.

Após a etapa de mapeamento estiver concluída, o *framework*, realiza um pré-processamento sobre os pares intermediários antes de submetê-los à função de redução. O *framework* ordena o conjunto de pares pela chave (etapa de ordenação), e agrupa todos os valores que possuem *a mesma chave* em apenas um par (etapa de combinação). Dessa forma, é obtido um novo conjunto intermediário de pares $\langle chave, list\,(valor)\rangle$, que são repassados à função de redução. A função de redução recebe cada chave com sua respectiva lista de valores, e realiza uma determinada operação sobre tais valores, por exemplo, somando-os.

A seguir, são apresentados os exemplos da contagem de palavras e da construção de um índice invertido dado um conjunto de documentos.

2.1 Contagem de palavras

Um exemplo simples do uso do MapReduce trata-se do problema em contar as ocorrências de palavras em um grande conjunto de documentos. As funções de mapeamento (*map*) e redução (*reduce*) para este problema são apresentadas na Figura 1.

```
map(String chave, String valor):
    Tokens tokens = Tokenize(valor);
    para cada token t em tokens:
        CriaIntermediario(t, "1");
```

```
reduce(String chave, Iterador valores):
    int resultado = 0;
    para cada valor v em valores:
        resultado += (int)v;
    retorne(chave, resultado);
```

Figura 1: Funções de mapeamento e redução da contagem de palavras

A função de mapeamento é invocada pelo *framework*, que submete à ela os valores extraídos de um documento. Nessa primeira fase, a *chave* pode ser, por exemplo, o nome do documento, e o *valor* pode ser uma determinada linha de texto extraída do respectivo documento. Dessa forma, a função de mapeamento gera *tokens* da linha do documento, onde cada *token* corresponde a uma palavra. Em seguida, a função associa cada *token* ao valor "1", criando os pares intermediários $\langle token,$ "1"$\rangle$.

Os pares intermediários são repassados ao *framework*, que agrupa todos os valores do mesmo *token*, criando os pares $\langle token, list\,(\text{"1"}, \text{"1"}, \text{"1"}, ...)\rangle$ ordenados pela chave. Após essa fase de pré-processamento, os pares intermediários são submetidos à função de redução, que recebe como parâmetro uma palavra (que possui o papel de chave) e a lista de valores agrupados da respectiva palavra. A função de redução simplesmente itera a lista e acumula todos os valores da chave na variável `resultado`, retornando um novo par $\langle chave, resultado\rangle$, que expressará a palavra associada ao seu número de ocorrências.

2.2 Geração de um índice invertido

Um índice invertido é uma estrutura de dados bastante comum em algoritmos de indexação, sendo composto de uma lista de termos que são relacionados a uma lista de documentos que os contém. A Figura 2 ilustra um exemplo para três palavras, onde o valor inteiro ao lado de cada documento representa o número de ocorrências da respectiva palavra no documento.

A construção do índice invertido de um conjunto de documentos pode ser expresso pelas funções de mapeamento e redução exibidas na Figura 3.

A função de mapeamento recebe como parâmetros um bloco de texto contido no documento cujo nome é especificado no primeiro parâmetro. Em sequência, a função extrai os *tokens* do texto, onde

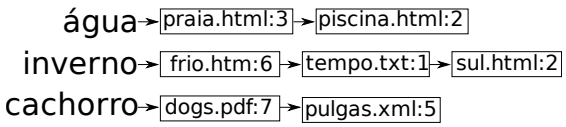

Figura 2: Exemplo de índice invertido

```
map(String nome_doc, String texto):
    Tokens tokens = Tokenize(texto);
    para cada token t em tokens:
        CriaIntermediario(t, nome_doc);
```

```
reduce(String palavra, Iterador nome_docs):
    para cada nome_doc em nome_docs:
        ocorrencias[nome_doc] += 1;

    list_ocorrencias = list();
    para cada oc em ocorrencias:
        list_ocorrencias.push([oc.indice,
                               oc.valor]);

    retorne(chave, list_ocorrencias);
```

Figura 3: Funções de mapeamento e redução do índice invertido

cada *token* corresponde a uma palavra, e relaciona-os com o nome do arquivo, criando assim os pares intermediários $\langle palavra, nome\_do\_arquivo \rangle$. Da mesma forma que no exemplo da contagem de palavras, o *framework* agrupa todos os nomes de documentos que contém as mesmas chaves (palavras), criando assim os pares $\langle palavra, list\,(nome\_doc1, nome\_doc1, nome\_doc2, ...) \rangle$.

A função de redução recebe cada palavra com sua respectiva lista de documentos, e conta quantas vezes cada documento aparece na lista, por meio do *array* associativo `ocorrencias`. Essa contagem corresponde ao número de ocorrências da palavra em cada documento. Em seguida, é criada uma lista de documentos (`list_ocorrencias`), que conterá as tuplas "[nome_documento, numero_ocorrencias]", para que, ao final, a função de redução devolva a palavra e os documentos onde ela ocorre.

3 Plataforma Hadoop Core

A plataforma Hadoop Core é um *framework* que permite o desenvolvimento de aplicações distribuídas no modelo MapReduce, onde a principal característica de tais aplicações é a necessidade em processar grandes volumes de dados (por exemplo, na ordem de petabytes). As aplicações são desenvolvidas na linguagem Java, sendo o *framework* responsável por tratar questões de escalabilidade, tolerância a falhas, balanceamento de carga e distribuição de dados da aplicação.

O Hadoop Core trabalha com um sistema de arquivos distribuído próprio, chamado *Hadoop Distributed File System* (HDFS). O HDFS é um sistema de arquivos que foi desenvolvido para ser executado em hardware não-confiável, como em *clusters* formados por computadores de uso geral. O HDFS distribui os arquivos em blocos de dados entre os nodos, os quais mantém os blocos em seus meios de armazenamento locais. Além disso, o HDFS distribui cópias idênticas dos blocos de arquivos em outros nodos, como meio de recuperar blocos de dados no caso de falha de um determinado nodo.

3.1 Arquitetura

O Hadoop Core é orientado a uma arquitetura mestre/escravo, e possui, basicamente, quatro componentes principais: o *JobTracker*, o *NameNode*, o *TaskTracker* e o *DataNode*. O *JobTracker* e o *NameNode* fazem o papel dos mestres, enquanto que o *TaskTracker* e o *DataNode* são os escravos da aplicação. Embora seja alocado um *TaskTracker* e um *DataNode* para cada nodo disponível no agregado, existem apenas uma instância do *JobTracker* e do *NameNode*.

3.1.1 NameNode e DataNode

O NameNode e o DataNode são os componentes responsáveis por manter os dados do sistema de arquivos HDFS. O NameNode é o componente mestre que gerencia o espaço de nomes do sistema de arquivos e regula o acesso aos arquivos pelos clientes. Os DataNodes são executados em cada nodo do cluster, e têm a atribuição de gerenciar o armazenamento dos *blocos de dados* dos arquivos nos discos locais do nodo.

Internamente, os arquivos armazenados no HDFS são divididos em blocos de dados, que são atribuídos e mapeados aos DataNodes pelo NameNode. Operações de abertura, fechamento e renomeação de arquivos e diretórios são feitas apenas pelo NameNode. Operações de leitura e escrita em arquivos são feitas diretamente pelos DataNodes, isso é, quando um cliente necessita alterar um bloco de dados de um arquivo, não é necessário requisitar a operação ao NameNode. Além disso, os DataNodes são responsáveis por realizar operações de criação, deleção e replicação de blocos de arquivos, porém apenas sob orientação do NameNode [Borthakur, 2009].

3.1.2 JobTracker e TaskTracker

O *JobTracker* é o componente que faz o ponto de interação entre o usuário e o *framework* Hadoop Core. Os jobs MapReduce são submetidos ao JobTracker que os coloca em uma fila de espera, e são executados através de uma política de escalonamento *first-come/first-served*. Quando um job é escalonado à execução, o JobTracker conversa com o NameNode para determinar a localização dos dados necessários ao processamento, e submete as tarefas de *map* e *reduce* aos TaskTrackers que estejam nos nodos mais próximos aos blocos de dados (por exemplo, no mesmo *rack* onde os nodos estão instalados fisicamente), otimizando assim a disponibilidade dos dados que os jobs devem processar [Loughran, 2008].

Os TaskTrackers possuem um número pré-definido de *slots* de execução, que são ocupados pelas tarefas de *map* ou de *reduce*. Todas as tarefas alocadas em um TaskTracker são executadas em máquinas virtuais (JVM) diferentes para garantir que o TaskTracker não seja encerrado devido à uma falha da aplicação. Além disso, os TaskTrackers enviam constantemente *heartbeats* ao JobTracker, que pode alocar novos TaskTrackers para a re-execução da tarefa de um TaskTracker sinalizado como morto.

4 Programando no Hadoop Core

A API de programação do Hadoop Core é distribuída conjuntamente com o *framework*, que possui em torno de 41 MB. O desenvolvimento deste trabalho foi feito com a última versão disponível da plataforma (*22 April, 2009: release 0.20.0*).

Para desenvolver uma aplicação MapReduce no Hadoop Core, o programador deve, basicamente, implementar uma classe de mapeamento e uma classe de redução. Na classe principal (que contém o método `main`), são especificadas a entrada e a saída dos dados no HDFS, e quais são as classes responsáveis pelo mapeamento e pela redução.

Este trabalho desenvolveu a aplicação de contagem de palavras (WordCount) e criação do índice invertido (InvertedIndex), ambas com os códigos-fonte disponíveis. O código-fonte da aplicação WordCount foi anexado ao final deste relatório com o intuito de facilitar o acompanhamento das sub-sessões seguintes, que apresentam os detalhes de programação das classes de mapeamento (Anexo A), redução (Anexo B), e do corpo do programa principal (Anexo C).

4.1 Classe de Mapeamento

A classe de mapeamento da aplicação deve estender a classe `Mapper` (`org.apache.hadoop.mapreduce.Mapper`), cujos métodos são listados abaixo.

Classe `Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT>`

map(KEYIN chave, VALUEIN valor, Context context)

run(Context context)

setup(Context context)

cleanup(Context context)

Em `KEYIN` e `VALUEIN` devem ser especificados os tipos de dados da chave e do valor de entrada. Da mesma forma, em `KEYOUT` e `VALUEOUT`, devem ser especificados os tipos de dados dos pares intermediários que o método de mapeamento irá gerar. O programador deverá especificar o procedimento de mapeamento sobrecarregando o método `map`.

Da mesma forma que o método `map`, os demais métodos `run`, `setup` e `cleanup` também podem ser sobrecarregados pela aplicação. Os métodos `run` e `cleanup` são chamados antes e após a execução do mapeamento. Portanto, tais métodos podem ser utilizados caso seja necessário realizar qualquer tipo de inicialização/finalização.

O método `run` é invocado pelo *framework*. Este método é responsável por chamar os métodos `setup` e `cleanup` antes e após as chamadas ao método `map`. O método `run` obtém cada par $\langle chave, valor \rangle$ do objeto *context*, e os submete à função `map`. A função `map`, por sua vez, gera os pares intermediários, emitindo-os ao *framework* por meio do método `context.write(KEYOUT chave, VALUEOUT valor)`.

4.2 Classe de Redução

A classe de redução deve estender a classe `Reducer` (`org.apache.hadoop.mapreduce.Reducer`), cujos métodos são especificados abaixo, sendo bastante semelhantes aos da classe `Mapper`.

Classe `Reducer<KEYIN, VALUEIN, KEYOUT, VALUEOUT>`

reduce(KEYIN chave, Iterable<VALUEIN> valores, Context context)

run(Context context)

setup(Context context)

cleanup(Context context)

O método `reduce` deve ser sobrecarregado pela aplicação, e recebe todas as chaves intermediárias com seus respectivos conjuntos de valores (tipo `Iterable`). Tais conjuntos de valores são aqueles formados pela etapa de combinação feita pelo *framework*. Os métodos `run`, `setup` e `cleanup` possuem os mesmos propósitos dos métodos da classe `Mapper`.

Os pares finais criados pelo mapeamento são emitidos através do objeto `context`, por meio do método `write(KEYOUT chave, VALUEOUT valor)`.

4.3 Classe principal do programa

Uma vez que as classes de mapeamento e redução estejam definidas, a classe principal do programa (que contém o método `main`) deve especificar ao *framework* quais as classes responsáveis pelo mapeamento e redução dos valores. Dessa forma, o *framework* tem condições de alocar os respectivos objetos aos TaskTrackers.

As classes *Mapper* e *Reducer* da aplicação devem ser especificadas por meio do objeto `Job`, cujos métodos principais são exibidos abaixo.

Classe `Job`

setJarByClass(Class classe)

setMapperClass(Class classe)

setReducerClass(Class classe)

setMapOutputKeyClass(Class classe)

setMapOutputValueClass(Class classe)

setOutputKeyClass(Class classe)

setOutputValueClass(Class classe)

waitForCompletion()

submit()

O objeto `Job` deve ser instanciado passando ao construtor um objeto da classe `Configuration`, e um nome para a aplicação. Através do objeto da classe `Configuration`, o *framework* tem acesso aos parâmetros de configuração da plataforma, contidos dentro do diretório `conf` do Hadoop Core.

No método `setJarByClass` deve ser especificada a classe do programa principal; nos métodos `setMapperClass` e `setReducerClass` devem ser especificadas as classes *Mapper* e *Reducer* da aplicação; através dos métodos `setMapOutputKeyClass` e `setMapOutputValueClass` os tipos de dados da chave e do valor dos pares gerados pelo objeto de mapeamento devem ser especificados. Os tipos de dados dos pares finais resultantes são especificados por meio dos métodos `setOutputKeyClass` e `setOutputValueClass`.

Além da configuração do `Job`, deve-se especificar a entrada e a saída de dados por meio das classes `FileInputFormat` e `FileOutputFormat`, respectivamente. O caminho de entrada é especificado através do método de classe *addInputPath*, que adiciona um caminho a uma lista de diretórios onde os dados serão buscados para um determinado job, que também deve ser passado como parâmetro. O diretório onde os dados resultantes do job serão gravados deve ser especificado através do método *setOutputPath*.

Outra forma de construção da classe principal, que é exemplificada na aplicação InvertedIndex, é a de estender a classe `Configured` e implementar a interface `Tool`. A classe `Configured`, da mesma forma que o objeto `Configuration`, possibilita acesso ao *framework* às configurações da plataforma. Através da interface `Tool`, a classe deve implementar o método `run`, onde todo o código do programa principal deve estar contido, que é invocado através da classe `ToolRunner` chamada no método `main`. A razão em executar a aplicação por meio da classe `ToolRunner` é de que essa classe é capaz de tratar argumentos genéricos da aplicação como, por exemplo, especificar um JobTracker ou um NameNode diferente dos padrões [Foundation, 2008].

A submissão do job a execução deve ser feita por meio do método `waitForCompletion`, que bloqueia o programa até que a execução termine. Como uma alternativa, o método `submit` pode ser utilizado para submeter o job à execução, não causando o bloqueio do programa.

4.4 Exemplos de aplicações

Para exemplificar o uso da API da plataforma Hadoop, foram desenvolvidas as aplicações de contagem de palavras (WordCount) e de construção do índice invertido de um conjunto de documentos (InvertedIndex). O código-fonte da aplicação WordCount é apresentado nos Anexos A, B e C.

Os projetos do ambiente NetBeans de ambas as aplicações são distribuídos em conjunto com o presente relatório sob licensa GNU GPLv3. Além dos projetos das aplicações, no diretório *exemplo_entrada* contém um conjunto de documentos HTML extraídos de http://www.gnu.org/philosophy que pode ser utilizado como exemplo de entrada para ambas aplicações.

4.5 Executando a aplicação

O Hadoop Core admite três tipos de instalação: *não-distribuída*, *pseudo-distribuída* e *distribuída*. Este trabalho realizou os testes utilizando uma instalação *pseudo-distribuída*, cujos passos de configuração podem ser encontrados no link http://hadoop.apache.org/core/docs/r0.20.0/quickstart.html. O sistema operacional utilizado foi o Ubuntu Linux 9.04.

Considerando que o *framework* tenha sido devidamente inicializado, a execução do programa paralelo será exemplificada por meio da aplicação WordCount. Primeiro, deve-se gravar no HDFS os arquivos de entrada do programa. Por exemplo, o comando abaixo grava todo o conteúdo do diretório *paginas_html* no diretório *entrada* do sistema de arquivos distribuído. Se o diretório *entrada* não existir, ele é criado automaticamente.

```
$ hadoop dfs -put paginas_html/ entrada/
```

O conteúdo do sistema de arquivos pode ser listado por meio do comando

```
$ hadoop dfs -ls
```

Considerando que a aplicação já tenha sido compilada e esteja contida no `Jar` WordCount.jar, a execução é feita por meio da linha de comando que segue.

```
$ hadoop jar WordCount.jar WordCount entrada/ saida/
```

O programa irá carregar todos os arquivos contidos no diretório *entrada*, e o resultado da computação será gravado no HDFS dentro do diretório *saida*. O diretório *saida* não pode existir no HDFS, caso contrário o *framework* irá gerar uma exceção.

Uma vez que a execução tenha sido disparada, a plataforma possibilita monitorar o andamento da execução das tarefas. Considerando uma instalação pseudo-distribuída padrão, todos os jobs submetidos são relacionados no endereço http://localhost:50030/, onde é possível clicar no job que está sendo executado e verificar o andamento da execução. Após a execução ter sido concluída, o resultado pode ser obtido acessando o endereço http://localhost:50070/, que permite navegar no sistema de arquivos do Hadoop.

5 Considerações finais

Através da experimentação do modelo de programação MapReduce por meio da plataforma Hadoop Core, foi possível perceber que a plataforma agrega uma considerável abstração ao paralelismo do programa. Isso é, o programador é poupado de determinados aspectos da programação paralela, como balanceamento de carga, tolerância a falhas, escalonamento e distribuição dos dados. Além disso, a API da plataforma é relativamente simples e direta, o que facilita o desenvolvimento da aplicação.

Embora a plataforma traga uma abstração ao paralelismo, o desenvolvimento das aplicações no Hadoop Core se restringe ao modelo MapReduce. Ou seja, a aplicação *deve* ser expressa por meio de uma função de mapeamento e redução. Caso contrário, o modelo não se aplica.

Além disso, constatou-se que a plataforma Hadoop Core na sua versão 0.20.0, até a presente data (10 de julho de 2009), possui uma documentação um tanto inconsistente com relação à API de programação. A organização das classes nos pacotes do Hadoop foi alterada de forma significante da versão 0.19 para a 0.20, enquanto que os exemplos distribuídos conjuntamente com a plataforma, bem como os tutoriais na página do projeto, não foram atualizados para a nova versão da API. Exemplo disso é a aplicação WordCount, que foi re-escrita para a nova versão da API.

Referências

[Borthakur, 2009] Borthakur, D. (2009). Hdfs architecture. http://hadoop.apache.org/core/docs/current/hdfs_design.html.

[Dean and Ghemawat, 2008] Dean, J. and Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113.

[Foundation, 2008] Foundation, A. (2008). Class toolrunner. http://hadoop.apache.org/core/docs/r0.20.0/api/org/apache/hadoop/util/ToolRunner.html.

[Loughran, 2008] Loughran, S. (2008). Jobtracker. http://wiki.apache.org/hadoop/JobTracker.

A Classe Mapper – WordCount

```
// Classe de mapeamento: herda de Mapper
public class WordCountMapper extends Mapper<LongWritable, Text, Text, IntWritable> {

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    // O processamento do Mapper eh feito no metodo map, que deve ser sobrecarregado
    @Override
    public void map(LongWritable key, Text value, Context context)
                                        throws IOException, InterruptedException {
        // Extrai os tokens da linha de texto
        StringTokenizer tokenizer = new StringTokenizer(value.toString());
        // Faz o mapeamento (token, ''1'') para todos os tokens
        while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            context.write(word, one);
        }
    }
}
```

B Classe Reducer – WordCount

```
// Classe de mapeamento: herda de Reducer
public class WordCountReducer extends Reducer<Text, IntWritable, Text, IntWritable> {

    // O processamento do Reducer eh feito no metodo reduce
    // O metodo run passa para a funcao um token (key) e a lista
    //  de valores unitarios
    // A funcao reduce simplesmente soma os valores unitarios de cada
    //  palavra e cria o par <palavra, total_de_ocorrencias>
    @Override
    public void reduce(Text key, Iterable<IntWritable> values, Context context)
                        throws IOException, InterruptedException {
        int sum = 0;

        // Soma os valores unitarios da palavra
        for (IntWritable value : values) {
            sum += value.get();
        }
        // Cria o par
        context.write(key, new IntWritable(sum));
    }
}
```

C Programa principal – WordCount

```
public class WordCount {
    public static void main(String[] args) throws Exception {
        if (args.length < 2) {
            System.err.println("Uso: WordCount <entrada> <saida>");
            System.exit(2);
        }

        // Carrega os arquivos de configuracao da pasta conf
        Configuration conf = new Configuration();

        // Job controla parametros da execucao da aplicacao paralela
        Job job = new Job(conf, "Word Count");

        // Seta as classes da aplicacao, do mapper e do reducer
        job.setJarByClass(WordCount.class);
        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(WordCountReducer.class);

        // Seta os tipos de saida do mapper
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(IntWritable.class);

        // Seta os tipos de saida do reducer
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        // Caminho dos dados de entrada e saida no HDFS
        FileInputFormat.addInputPath(job, new Path(args[1]));
        FileOutputFormat.setOutputPath(job, new Path(args[2]));

        // Submete o job a execucao e bloqueia ate que ele termine
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```