UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

HADOOP MAPREDUCE TOLERANTE A FALTAS BIZANTINAS

Pedro Alexandre Reis Sá da Costa

MESTRADO EM INFORMÁTICA

2011

UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

HADOOP MAPREDUCE TOLERANTE A FALTAS BIZANTINAS

Pedro Alexandre Reis Sá da Costa

DISSERTAÇÃO

Projecto orientado pelo Prof. Doutor Marcelo Pasin

MESTRADO EM INFORMÁTICA

2011

Agradecimentos

Ao Professor Doutor Marcelo Pasin que, desde o início, me acompanhou juntamente com o Professor Miguel Correia, e que posteriormente tornou-se no meu orientador. Também me acompanhou ao longo do meu trabalho através das discussões de ideias, sugestões e na grande ajuda que me deu para a construção do protótipo. Também agradeço pela disponibilidade a responder-me às minhas dúvidas e pelas sugestões que foram extremamente importantes para a elaboração do artigo científico e da tese. Por me ter convidado e acolhido como membro da equipa dos Navigators. É um sonho que tenho perseguido há algum tempo, mas só agora tive a oportunidade, e por isso, estou-lhe extremamente grato.

Ao Professor Doutor Miguel Pupo Correia por ter sido a primeira pessoa a apresentar-me o campo de estudo do tema de trabalho e me ter apresentado o meu orientador. Por sempre me ter acompanhado no decurso do meu trabalho através das discussões de ideias e sugestões que foram extremamente importantes para a elaboração e construção do protótipo. Agradeço pela grande ajuda, contribuição e conselhos que me deu para a construção do artigo científico.

Ao Professor Doutor Alysson Neves Bessani pela ajuda que deu na revisão do artigo e pela partilha do conhecimento.

Ao Vinícius Vielmo Cogo pela ajuda e disponibilidade que teve comigo em explicar-me como o Grid'5000 e o *cluster* da faculdade funciona.

Este trabalho foi parcialmente financiado pela FCT e EGIDE (Programa PESSOA) através do projecto FTH-Grid, pelo EC FP7 através do projecto TCLOUDS (ICT-257243), pela FCT através do Programa Multi-anual e o projecto CloudFIT, e os fundos do Programa PIDDAC do INESC-ID. Agradeço calorosamente aos colegas do projecto FTH-Grid pelas várias discussões sobre os tópicos: Luciana Arantes, Vinícius Cogo, Olivier Marin, Pierre Sens, Fabrício Silva, Julien Sopena.

Dedicatória.

Dedico o meu trabalho aos meus pais, porque me têm sempre acompanhado ao longo da minha vida, dando-me sempre o suporte para eu poder viver. Este esforço é extraordinário e difícil de alguém conseguir equipará-lo. Ao mesmo tempo, espero conseguir mostrar-lhes que o esforço que têm feito nunca foi em vão e o facto de ser feliz é a melhor retribuição que lhes posso dar.

Dedico o meu trabalho aos meus avós, por serem fantásticos comigo e por me darem memórias felizes cada vez que estou com eles.

Dedico o meu trabalho ao meu irmão e à sua família que, apesar de termos percursos e personalidades diferentes, são fantásticos como pessoas. A ligação invisível que existe com o meu irmão nunca é referida e ainda bem, mas está lá e espero que continue através da humildade, compreensão e capacidade de gestão pessoal, porque no fundo, o que interessa é o respeito pelo próximo.

Dedico o meu trabalho à Tia Teresa e à Tia Alice, que não estando já entre nós, fazem muita falta. Ao Tio Humberto e ao resto da família, como o pequeno Lourenço, pelas férias fantásticas de Verão no Parque de Campismo e sardinhadas depois da praia. Ao Tio Domingos, ao Senhor Coelho, à Luísa e ao Sérgio por serem pessoa simpáticas.

Dedico o meu trabalho aos meus cães, os dois chamados Vasco, que já não estando entre nós, e deixando algumas marcas como recordação, gosto bastante deles. Dedico o meu trabalho à minha cadela Ema, que sem saber, é um pilar extremamente importante na família.

À Ana Sofia Santos porque pude mostrar-lhe que é possível fazermos aquilo que gostamos, mesmo quando o mundo inteiro nos nega. Para viver basta sermos honestos, humildes e trabalhadores pois tudo o que tem que acontecer, acontece. A vida é extraordinária, ela não se atrasa, nem se adianta um segundo. Tudo acontece à hora certa.

Aos meus colegas André Nogueira, Bruno Vavala, João Antunes, Mônica Dixit, Patrícia Gonçalves e Vinícius Cogo pelo companheirismo e simpatia que são, tornando os dias de trabalho agradáveis.

Dedico o meu trabalho aos Professores Doutores Paulo Jorge Esteves Veríssimo, Marcelo Pasin, Miguel Pupo Correia, Nuno Ferreira Neves e António Casimiro, por serem as pessoas mais importantes na minha formação e área de trabalho, e espero ter a honra de poder mostrar-lhes através do meu trabalho que a transmissão do conhecimento foi bem passada e que um dia consiga contribuir de forma tão relevante como fazem.

Dedico o meu trabalho aos restantes docentes do Departamento de Informática, e à Faculdade de Ciências, por existir como instituição e lutar por um trabalho cientificamente reconhecido. De forma indirecta acolheram-me e oferecem-me uma experiência de vida entusiasmante. O resultado deste trabalho é proveniente do conhecimento que a instituição me transmitiu e da minha capacidade, do qual procurei tirar o melhor proveito para servi-la.

À ciência que me atrai como o exercício lógico do pensamento humano.

E por último, às pessoas que por situações particulares me dificultaram o caminho, mas silenciosamente e sem saberem, induziram-me no caminho que eu considero correcto e assim consegui perceber melhor o mundo, valorizar a vida e as pessoas, e crescer.

Deixo à vossa consideração este trabalho.

"Em nome dos sonhos, e mais além."

Resumo

O MapReduce é frequentemente usado para executar tarefas críticas, tais como análise de dados científicos. No entanto, evidências na literatura mostram que as faltas ocorrem de forma arbitrária e podem corromper os dados. O Hadoop MapReduce está preparado para tolerar faltas acidentais, mas não tolera faltas arbitrárias ou Bizantinas. Neste trabalho apresenta-se um protótipo do Hadoop MapReduce Tolerante a Faltas Bizantinas(BFT). Uma avaliação experimental mostra que a execução de um trabalho com o algoritmo implementado usa o dobro dos recursos do Hadoop original, em vez de mais 3 ou 4 vezes, como seria alcançado com uma aplicação directa dos paradigmas comuns a tolerância a faltas Bizantinas. Acredita-se que este custo seja aceitável para aplicações críticas que requerem este nível de tolerância a faltas.

Palavras-chave: Hadoop MapReduce, Tolerância a Faltas Bizantinas, faltas arbitrárias, replicação

Abstract

MapReduce is often used to run critical jobs such as scientific data analysis. However, evidence in the literature shows that arbitrary faults do occur and can probably corrupt the results of MapReduce jobs. MapReduce runtimes like Hadoop tolerate crash faults, but not arbitrary or Byzantine faults. In this work, it is presented a MapReduce algorithm and prototype that tolerate these faults. An experimental evaluation shows that the execution of a job with the implemented algorithm uses twice the resources of the original Hadoop, instead of the 3 or 4 times more that would be achieved with the direct application of common Byzantine fault-tolerance paradigms. It is believed that this cost is acceptable for critical applications that require that level of fault tolerance.

Keywords: Hadoop MapReduce, Byzantine Fault-Tolerance, arbitrary faults, replication

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 MapReduce

O MapReduce é um modelo de programação e uma plataforma de software usada para escrever aplicações que processam grandes quantidades de dados em paralelo. O modelo de programação procura dividir um problema em subproblemas, ao qual, para cada um deles, vai encontrar soluções e combiná-las no final da execução.

Um programa segundo o modelo MapReduce tem duas funções: *map* e *reduce*. A execução deste programa faz-se em um *job*, que recebe um ficheiro de entrada e gera um ficheiro de saída. O *job* processa os dados com tarefas elementares, do tipo *map* ou do tipo *reduce*, que executam as respectivas funções do programa. O ficheiro de entrada é dividido em várias partes pequenas, cada uma delas é processada por uma tarefa de *map*, gerando mapas (conjuntos chave-valor). Os mapas gerados são concatenados, ordenados por chave e novamente divididos em partes com a mesma chave. Estas novas partes são entregues a tarefas do tipo *reduce*, que reduzem a informação recebida, geram e concatenam o resultado final num ficheiro de saída, por ordem de chave.

O modelo de programação MapReduce é uma abordagem promissora para computação científica[16, 20] e é usado por várias empresas para a extracção, pesquisa e análise de dados, geração de estatísticas e outras funcionalidades que se limitem ao campo da análise sintáctica de informação.

A nível de software, o MapReduce é uma plataforma desenvolvida pela Google para o processamento de grandes quantidades de dados[14]. A implementação da Google não está abertamente disponível, mas uma versão *open source* chamada Hadoop [1] [43] é usada por muitas empresas, como a Amazon, eBay, Facebook, IBM, LinkedIn, RackSpace, Twitter e Yahoo!, nos seus modelos de negócio. Um último argumento em favor da importância do MapReduce é que versões comerciais estão aparecendo, como MapReduce Windows Azure e Amazon Elastic MapReduce.

[1] Hadoop é um projecto *open source* da Apache com muitos componentes. Usa-se o termo Hadoop neste documento para se referir à aplicação MapReduce (http://wiki.apache.org/hadoop/PoweredBy)

1.2 Motivação

O MapReduce foi projectado para ser tolerante a faltas, pois em ambientes da escala de milhares de computadores e centenas de outros dispositivos como *switches* de rede, *routers* e *power units*, as falhas dos componentes são frequentes. Por exemplo, no primeiro ano de um *cluster* na Google, 1000 máquinas individuais falharam e aconteceram centenas de falhas nos discos rígidos [13]. O MapReduce da Google e do Hadoop toleram falhas nas tarefas de map e reduce. Se uma dessas tarefas falhar antes de terminarem, a falha é detectada e uma nova instância da mesma tarefa é criada. Além disso, os dados são armazenados em disco juntamente com *checksums*, para detectar facilmente se estão corruptos [18, 30, 43, 40].

Embora seja crucial tolerar falhas nas tarefas e dados corruptos, outras faltas que podem afectar a *exactidão dos resultados* do MapReduce são conhecidas e irão acontecer mais no futuro[38] devido à popularidade da plataforma. Um estudo recente dos erros DRAM num grande número de servidores dos DataCenters da Google concluíram que esses erros são mais frequentes do que se pensava, com mais de 8% de DIMMs afectados anualmente por erros, mesmo estando protegidos por códigos de correcção de erros [39]. Um estudo da Microsoft em 1 milhão de PCs de consumo mostra que as faltas no CPU e chipset também são frequentes [32].

Os mecanismos de tolerância a faltas existentes no MapReduce não permite que a plataforma seja capaz de lidar com faltas arbitrárias, ou faltas Bizantinas [3] (não se considera faltas maliciosas). Estas faltas não podem ser detectadas usando *checksums*, o que significa que, muitas vezes, as tarefas terminam com sucesso, embora tenham gerado dados corruptos. Estes erros apenas são detectados ao se correr as tarefas várias vezes, e comparar-se os diferentes resultados. Esta ideia básica foi proposta no contexto de *volunteer computing* para tolerar voluntários maliciosos que retornam resultados falsos das tarefas que deveriam executar [36]. Esse trabalho, no entanto, considerado como uma aplicação *bag-of-tasks*, é mais simples do que a plataforma MapReduce. Uma solução semelhante, mas mais genérica é a abordagem de máquina de estado, no qual um conjunto de programas são executados em paralelo por diferentes servidores, que executam comandos na mesma ordem [37]. Esta abordagem, no entanto, não é directamente aplicável à replicação de tarefas no MapReduce, apenas nos serviços que seguem o modelo cliente-servidor (por exemplo, um serviço de nomes ou um servidor de ficheiros). Uma solução ingénua seria executar duas vezes o mesmo exemplo e comparar os resultados, mas o custo é excessivo nos caso de falhas.

A principal motivação deste trabalho é implementar um algoritmo MapReduce tolerante a faltas Bizantinas, já que a plataforma oficial não consegue lidar com este tipo de faltas, e o trabalho científico realizado não lida com faltas arbitrárias da mesma forma como este algoritmo faz.

1.3 Objectivos

O Hadoop MapReduce está preparado para tolerar faltas acidentais, mas não tolera faltas arbitrárias ou Bizantinas. Neste trabalho apresenta-se um protótipo do Hadoop MapReduce Tolerante a Faltas Bizantinas (BFT). Este sistema tolera faltas que corrompem os resultados intermediários produzidos pelas tarefas, tais como os erros/falhas da DRAM e do CPU mencionadas na secção 1.2.

Esta solução é mais cara a nível do tempo de execução, de memória e espaço em disco, do que usar a versão oficial do MapReduce. Uma configuração típica exigirá que cada tarefa seja executada pelo menos duas vezes, o que é um *overhead* considerável em termos de recursos utilizados e, possivelmente, de tempo de execução. No entanto, acreditamos que este custo é aceitável para aplicações críticas que exigem um alto grau de certeza sobre a validade dos dados. Um grande conjunto de aplicações de computação científica cairá nesta categoria [16, 20].

1.4 Contribuições

O MapReduce BFT segue a abordagem da execução de trabalho replicado, à semelhança de muitos sistemas tolerante a faltas Bizantinas. O desafio é fazê-lo *de forma eficiente*, por exemplo, executando apenas 2 cópias de cada tarefa, quando não há faltas. Observe que, por exemplo, a abordagem de máquina de estados requer $3f + 1$ réplicas para tolerar, no máximo, f faltas, o que dá um mínimo de 4 cópias para cada tarefa [8, 9]. Usa-se vários mecanismos para minimizar tanto o número de cópias de tarefas a executar e o tempo necessário para executá-las. No caso de haver uma falta, o custo da nossa solução está perto do custo de se executar 2 vezes, em vez de 3 vezes como a solução ingénua proposta acima. Evidentemente, se houver mais de uma falta, a diferença terá de ser maior.

A principal contribuição do trabalho é a criação e implementação de um algoritmo capaz de tolerar faltas arbitrárias no Hadoop MapReduce. Tirou-se medidas de desempenho da versão, usando a ferramenta de benchmarking Gridmix.[2] Os resultados destes testes confirmaram que é possível executar o Hadoop MapReduce BFT, gastando o dobro do tempo total de execução e de CPU que a versão oficial.

1.5 Estrutura do documento

O capítulo 1 apresenta as razões de escolha deste tema de trabalho, a motivação e a as contribuições feitas. O capítulo 2 apresenta a plataforma Hadoop MapReduce e os componentes que a constitui. No capítulo 3 descreve-se o novo algoritmo e apresenta-se o protótipo. O capítulo 4 descreve os detalhes de implementação do protótipo. O capítulo

[2] http://hadoop.apache.org/mapreduce/docs/current/gridmix.html

5 apresenta os resultados dos testes realizados na plataforma oficial e na versão BFT, comparando os valores. No capítulo 6 descreve-se os pontos a implementar no futuro para melhorar o protótipo. Por último, no capítulo 7 apresenta-se uma conclusão sobre o trabalho.

Capítulo 2

Hadoop e Tolerância a Faltas Bizantinas

O modelo de programação MapReduce é um paradigma popular usado por bastantes companhias e tem sido sujeito a muito estudo. Existem empresas que desenvolvem a sua própria implementação customizada ao negócio, ou existem implementações próprias do modelo de programação.

Uma área em que um trabalho considerável tem sido feito, consiste em adaptar o MapReduce para um bom desempenho em diferentes ambientes e aplicações, tais como sistemas de multi-core e multi-processadores (Phoenix system) [35], ambientes heterogéneos como o Amazon EC2 [44], ambientes *peer-to-peer* [27], ambientes com *high-latency eventual-consistent* como o Windows Azure (sistema Azure MapReduce) [20], aplicações interactivas (sistema Twister) [15], aplicações que usam intensivamente a memória e o CPU (sistema LEMO-MR) [17]. Outra tendência importante é a pesquisa sobre o uso MapReduce para a computação científica, por exemplo, para a análise de dados da física de alta energia (*High Energy Physics Theory*) e *cluster* Kmeans [16], e para a geração de modelos digitais de elevação [23]. O MapReduce mostra a sua potencialidade em casos de processamento de grandes quantidades de dados, seja ela na análise ou em cálculos da informação. O MapReduce consegue analisar rapidamente terabytes ou petabytes de informação. Vários sistemas são semelhantes ao MapReduce, no sentido de que fornecem um modelo de programação para o processamento de grandes conjuntos de dados, mas que permitem interacções mais complexas e/ou fornecem um nível maior de abstracção: Dryad [21], Pig Latin [33], Nephele [4]. Todos estes trabalhos mostram a importância do modelo de programação MapReduce, mas do ponto de vista de tolerância a faltas, que é o relevante neste trabalho, eles não adicionam aos mecanismos existentes na plataforma esta nova funcionalidade.

Tolerância a faltas arbitrárias é uma tendência existente há muito tempo no estudo de tolerância a faltas. Mecanismos de votação para mascarar faltas Bizantinas em sistemas distribuídos foram introduzidos no início de 1980 [34, 25]. A abordagem da máquina de estado é uma solução genérica para fazer um serviço tolerante a faltas Bizantinas [37]. Tem-se mostrado que é prático implementar sistemas tolerantes a faltas Bizantinas [8]

e uma longa linha de trabalho tem aparecido, incluindo bibliotecas como o UpRight [9] e EBAWA [42]. O uso da máquina de estados não é o adequado para tornar o MapReduce tolerante a faltas Bizantinas. Teria-se que replicar a execução do MapReduce num conjuntos de servidores, mas o custo seria muito alto.

Sistemas de quórum Bizantino têm sido usado para implementar armazenamento de dados concorrentes [26, 29], mesmo em *cloud* [5]. Embora as técnicas de votação têm algo em comum com o que este trabalho implementou, essas soluções não podem ser usados para implementar MapReduce BFT porque não é um serviço de armazenamento, mas um sistema que faz processamento.

Muito recentemente, aparecer um trabalho semelhante em *volunteering computing* para aplicações MapReduce [31]. A solução é baseada em votação. As principais diferenças é que o trabalho foca-se em ambientes diferentes (*volunteering computing*) e não se preocupa em reduzir o custo e melhorar o desempenho, por isso não introduz qualquer uma das optimizações que são o núcleo do trabalho. Esse *paper* também apresenta um modelo probabilístico do algoritmo que permite avaliar a probabilidade de obter um resultado errado, algo que não se apresenta aqui.

O problema de tolerar faltas em programas paralelos executados em máquinas paralelas não fiáveis foi estudada há muito tempo por Kedem et al [22]. No entanto, eles propuseram uma solução baseada em auditar passos intermédios da computação para detectar faltas. Neste trabalho assume-se que não é prático detectar falhas arbitrárias na execução de programas aleatórios criados pelos utilizadores, então comparando duas ou mais execuções de uma tarefa é a única possibilidade de detectar processamento faltoso.

2.1 Tolerância a faltas Bizantinas

Os sistemas computacionais consistem numa variedade de *hardware* e *software* sujeito a eventuais falhas. Em muitos sistemas, tais falhas pode levar à perturbação do serviço. Os sistemas desenhados para serem tolerantes a faltas, perante falhas dos componentes, são capazes de continuar a oferecer um serviço fiável. Para se lidar com faltas, o mecanismo de redundância é usado para mascarar as falhas que alguns componentes do sistema sofreram [12].

A tolerância a faltas Bizantinas é uma especialização da área de tolerância a faltas, que foi inspirada a partir do problema dos Generais Bizantinos [24, 41] definido em 1980. Até 1999, as soluções apresentadas para tornar um sistema tolerante a faltas Bizantinas eram pouco praticáveis. Em 1999, Miguel Castro e Barbara Liskov sugeriram uma máquina de estados replicada capaz de lidar com faltas arbitrárias [7] e que acabou por contribuir bastante para a continuação da pesquisa a sistemas tolerante a faltas Bizantinas. Lidar com faltas arbitrárias é difícil, porque podem existir processos faltosos que mentem, ou que realizam conluio [24]. A forma de evitar o conluio é através da obtenção do consenso

realizado por um quórum. O problema do consenso tem sido especificado de várias formas, mas, dito de forma sumária, esta propriedade serve para garantir um acordo realizado por várias partes sobre uma certa operação. O consenso é usado para resolver problemas de computação, tais como a replicação numa máquina de estados, *group membership* e *atomic broadcast* [19].

A replicação de máquinas de estados (RME) é uma solução genérica para a concretização de serviços tolerante a faltas. Um serviço oferece um conjunto de operações que os clientes invocam através dos pedidos. Um serviço é concretizado mediante um conjunto de n servidores. Um parâmetro importante quando se fala de um serviço tolerante a intrusões, é a capacidade de suportar um limite de faltas. Em sistemas assíncronos tolerante a intrusões, este limite é imposto pelo protocolo de difusão atómica, cuja a resistência máxima é de $f = \lfloor \frac{n-1}{3} \rfloor$ em n servidores [10]. O algoritmo "Practical Byzantine Fault Tolerance" (PBFT), apresentado em [7], tem uma resistência máxima para sistemas assíncronos de $n \geq 3f + 1$ para n servidores.

A capacidade de resistência que se pretende do algoritmo PBFT, exige o uso de muitos recursos para tolerar poucas faltas. Por exemplo, se se quiser tolerar 2 faltas, são precisos 7 servidores. Para tolerar 3 faltas, são necessários 10 servidores. Estes algoritmos usam muitos recursos para tolerar poucas faltas, o que se trata de uma limitação. No mundo real, os recursos são finitos e custam muito dinheiro. Outras soluções têm sido criadas de forma a oferecer a mesma qualidade de serviço usando menos servidores. Existem soluções como a apresentada em [28] que oferece um serviço tolerante a faltas usando $2f + 1$ servidores. Esta solução parte do princípio que os pedidos já estão ordenados, e basta obter-se uma maioria, através de uma votação simples, para se considerar uma operação válida.

2.2 Hadoop MapReduce

O Hadoop MapReduce é uma implementação do modelo de programação baseado nas funções *map* e *reduce*, e um ambientes de execução que permite ao utilizador escrever facilmente aplicações que processam gigabytes ou petabytes de dados em paralelo em ambientes com um grande número de computadores, como sistema de *cluster* ou em *datacenters*.

Os utilizadores especificam as funções de *map* e de *reduce*: a primeira é usada para processar os dados de entrada e gerar pares chave-valor, o último é usado para concatenar os vários pares numa única chave e ordenar o conjunto de dados pela chave. Quando um exemplo é submetido com as funções definidas e a localização dos dados de entrada, os ficheiros de entrada têm que já existir no sistema de ficheiros do Hadoop (HDFS). A plataforma divide os dados de entrada em *splits* e atribui-os a cada tarefas de *map* (ver figura 2.1). Cada tarefa de *map* processa um *split* e gera um resultado chamado de

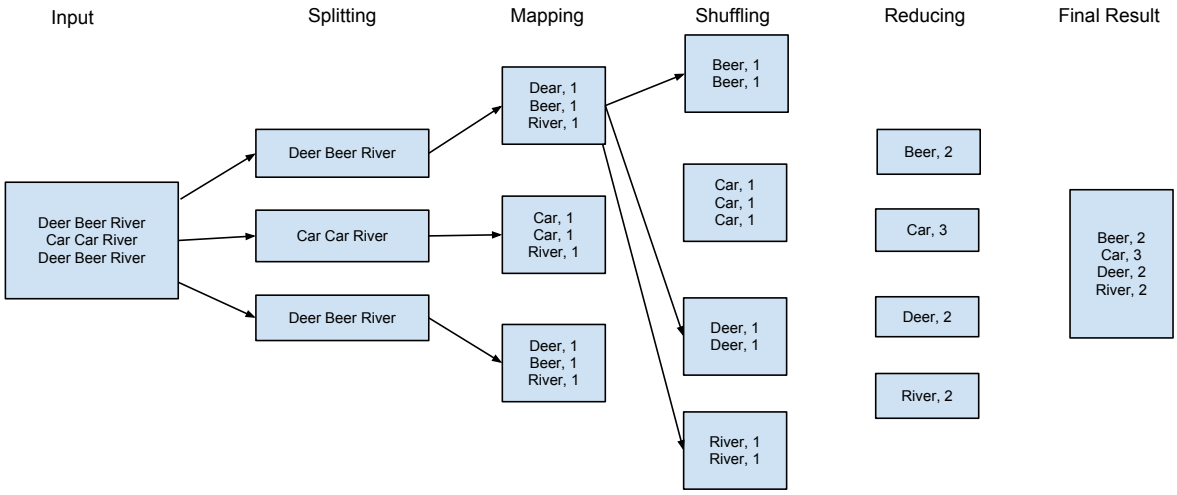

Figura 2.1: Exemplo do *WordCount*

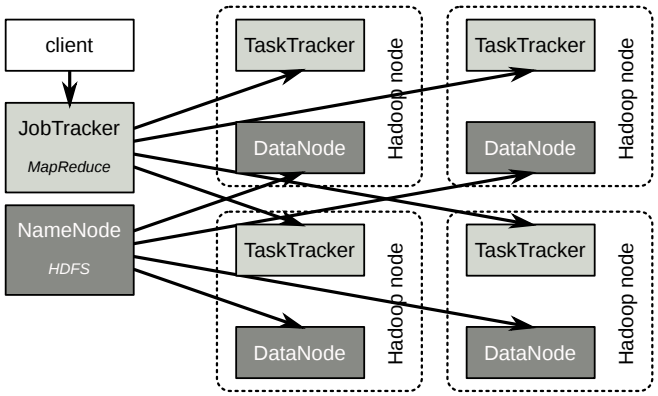

Figura 2.2: Arquitectura do Hadoop

MapOutput. Os dados do *MapOutput* são ordenados pela chave, particionados, e serão os dados de entrada para as tarefas de *reduce*. As tarefas de *reduce* processam estes dados e concatenam o resultado final num ficheiro de saída. De acordo com Dean e Ghemawat, é possível associar muitas tarefas do mundo real usando este modelo [14].

O Hadoop MapReduce é uma implementação do modelo de programação MapReduce realizado pela Apache e distribuído através da licença da Apache [43].

2.2.1 Como o MapReduce funciona

A plataforma é constituída por quatro tipos de componentes: *NameNode*, *DataNode*, *JobTracker* e *TaskTracker* (ver figura 2.2).

O *job* é uma unidade de trabalho que corresponde ao programa em execução que foi lançado pelo cliente. Cada *job*, ou unidade de trabalho, é constituído por um conjunto de tarefas de *map*, ou de *reduce*.

O *JobTracker* coordena a execução do *job*, o *TaskTracker* lança e gere a execução

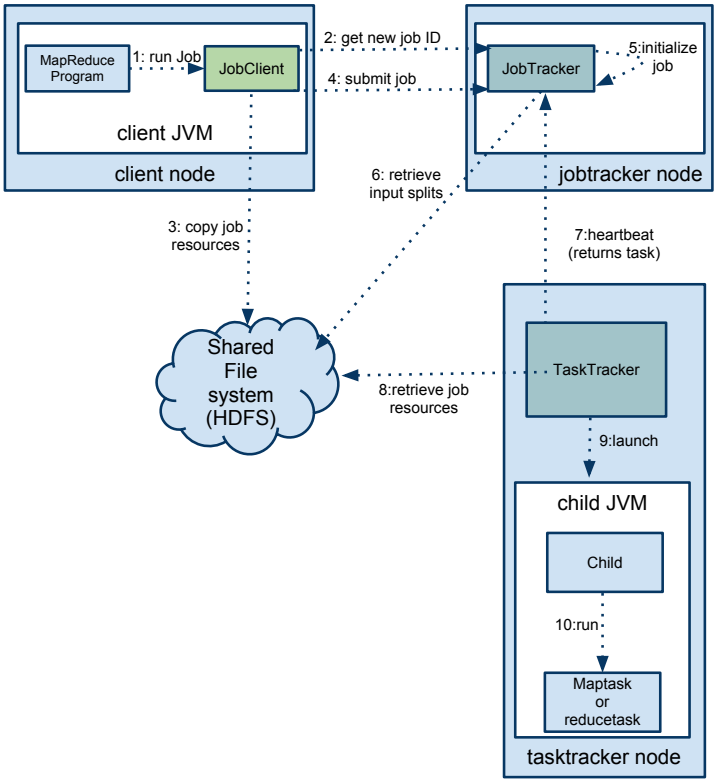

Figura 2.3: Sequência de passos que um exemplo passa

das tarefas, o *NameNode* contém a estrutura do sistema de ficheiros HDFS e o *DataNode* garante o acesso aos blocos de dados.

Todo o processo, desde a submissão do *job* até ao fim da sua execução, passa por uma sequência de fases. A figura 2.3 mostra um caso de uso genérico do progresso de um exemplo

Submissão do *job*: Durante a submissão do *job*, o *JobTracker* atribui um *JobID* (passo 2), sendo verificado se os dados de entrada existem (passo 3) e copiados todos os ficheiros necessários (ficheiros de configuração e *libraries*) para uma directoria temporária (passo 4). Este último passo é realizado por um tarefa especial, denominada de setup.

Inicialização do *job*: Após a submissão do *job*, o *JobScheduler* instancia as tarefas (passo 5).

Atribuição de tarefas Uma das funções do *JobTracker* é a distribuição do trabalho pelos *TaskTrackers*.

Um *TaskTracker* tem um número limitado de tarefas de *map* e de *reduce* que podem executar ao mesmo tempo. Ao distribuir tarefas, o *JobTracker* procura, primeiro, lançar as tarefas de *map*, antes das tarefas de *reduce*. O escalonador do *JobTracker*, componente

que distribui o trabalho, também tem noção da localização dos *splits* em relação ao *TaskTracker*. O *JobTracker* procura distribuir as tarefas pelos *TaskTrackers* mais próximos dos *DataNodes*. O Hadoop define as tarefas, conforme a localização dos *TaskTracker* em que ela corre, relativamente aos *DataNodes*, através da noção de `local`, `rack-local` e `remote`. No melhor caso, quando a tarefa corre no mesmo nó onde reside o *split*, a tarefa chama-se `local`. Se a tarefa correr no mesmo rack onde se encontra o *split*, a tarefa é `rack-local`. Em último caso, a tarefa é `remote`.

Execução de tarefas: Quando uma tarefa é atribuída a um *TaskTracker*, o próximo passo é lançá-la. O *TaskTracker* lança-a num novo processo. Durante a execução do processo filho, ele está constantemente a enviar o seu estado e progresso ao *TaskTracker*. Por sua vez, o *TaskTracker* reenvia a informação ao *JobTracker* através de um mecanismo que se denomina de *heartbeat*.

Progresso das tarefas: Para as tarefas de *map*, o progresso é calculado graças ao tamanho do *split* e da quantidade de dados já processados. Para as tarefas de *reduce*, o progresso é estimado pelas fases que passa (*Shuffle & Sort* e *Reduce*).

Job Completion: O *JobTracker* mantém uma tabela com o progresso de cada uma das tarefas, que é actualizada cada vez que um *TaskTracker* envia-lhe uma mensagem sobre o estado das suas tarefas. Quando o *JobTracker* é notificado do término da última tarefa, o *job* fica completo, as directorias temporárias, criadas na inicialização, são apagadas por uma tarefa especialmente denominada de *cleanup*. É de notar que cada *job* tem apenas uma tarefa de *setup* e uma de *cleanup* responsáveis por construir e remover a estrutura de directorias temporárias.

2.2.2 Fases de um *job*

Um *job* é constituído por várias tarefas. Uma total execução do *job* passa pela execução de todas as tarefas. Os vários momentos que uma unidade de trabalho passa denominam-se por fases. A execução das tarefas de *map* constitui a fase de *map*, a fase seguinte chama-se *shuffle & sort* e a execução das tarefas de *reduce* corresponde à fase de *reduce*.

Quando a tarefa de *map* começa a executar (fase de *map*), a primeira coisa que faz é obter os dados de entrada. De seguida, a tarefa executa a função `map` implementada pelo utilizador e guarda a sub-solução gerada localmente.

A fase seguinte é a fase *"shuffle & sort"*. Embora o nome desta fase possa ser enganador, ela é constituída por dois momentos:

Copy : O momento em que os dados de *MapOutput* são ordenados pela chave, e transferidos para o lado *reduce*.

Sort : O momento em que o *reduce* concatena o *MapOutput* e ordena os dados novamente pela chave.

Após as tarefas de *reduce* obterem o *MapOutput*, inicia-se a fase de *reduce*. Nesta fase, a tarefa executa a função `reduce` implementada pelo utilizador e obtém a solução final.

2.2.3 MapOutput

Todos os resultados gerados pelas tarefas de *map* são guardadas num disco local. Estes ficheiros chamam-se de *MapOutputs*. Cada *MapOutput* contém o resultado gerado pela função `map`. Um *MapOutput* é um único ficheiro constituído por pares chave-valor, que estão ordenados pela chave. Associado a um ficheiro de *MapOutput* encontra-se um ficheiro de índice que tem como função indicar o início e o fim de cada partição. O número de partições que cada *MapOutput* tem corresponde ao número de tarefas de *reduce* usadas.

Como cada nó contém uma partição por tarefa de *reduce*, elas vão buscar as partições correspondentes a cada nó. Estas partições obtidas são ordenada novamente pela chave, antes de passar para a fase de *reduce*.

2.2.4 Tolerância a faltas no Hadoop MapReduce

O Hadoop MapReduce contém mecanismos para tolerar faltas de paragem. A plataforma lida com estas faltas, relançando tarefas, ou criando tarefas especulativas. O relançamento da tarefa acontece quando a tarefa termina com o estado `FAILED`. O lançamento de uma tarefa especulativa acontece quando uma determinada tarefa está a demorar muito tempo para terminar.

Falhas nas tarefas

Quando um erro se manifesta numa das tarefas, há duas formas de lidar com ele. Se a tarefa falhar, ela é relançada pelo *TaskTracker*; se, em algum momento a tarefa deixar de responder ao *TaskTracker*, é lançada uma tarefa especulativa. Neste último caso, a primeira tarefa a terminar com sucesso, é considerada a correcta.

Uma das formas de lidar com falhas nas tarefas é através do uso de tarefas especulativas. Se uma tarefa deixar de comunicar com o *TaskTracker*, ou demorar muito tempo para terminar, é lançada uma tarefa especulativa. A tarefa especulativa é um clone da tarefa que substitui. Das duas, a primeira tarefa a terminar com sucesso será considerada a correcta. Esta forma de lidar com o problema tem como objectivo optimizar o tempo de execução do *job*. A noção de tarefas especulativas continuará a existir na nova versão.

Falhas no *TaskTracker*

O *TaskTracker* falha por um falta de paragem, ou por deixar de enviar *heartbeats* para o *JobTracker*. Se, durante 10 minutos, o *JobTracker* não receber um *heartbeat*, a tarefa é removida da *pool* de *TaskTrackers*. Todas as tarefas de *map* que lhe pertencem também serão relançadas. Se o número de falhas do *TaskTracker* for muito superior à média das falhas no *cluster*, o *JobTracker* põe-no numa lista negra e não lhe dá mais tarefas.

Falhas no *JobTracker*

A falha mais grave é quando o *JobTracker* falha. A plataforma não tem mecanismos para lidar com estas faltas. Se o *JobTracker* falhar, toda a plataforma fica comprometida.

Capítulo 3

Replicação de dados e tarefas

3.1 Hadoop MapReduce BFT

O Hadoop MapReduce BFT é um melhoramento da plataforma MapReduce que utiliza replicação e somas de controlo (ou *checksum*) como estratégia para se tornar tolerante a faltas Bizantinas. Na nova plataforma replica-se as tarefas de *map* e de *reduce* e usa-se somas de controlo para garantir a integridade dos dados.

Neste trabalho, assume-se que os clientes estão correctos. Se os clientes fossem maliciosos, não havia a necessidade de verificar a integridade dos resultados. Também se considera que o *JobTracker* não é faltoso, o que é a mesma assunção tomada pelo Hadoop. Contudo, esta consideração será removida em trabalho futuro. Mas, em respeito aos *TaskTrackers*, já não é tomada esta assunção em consideração. Os *TaskTrackers* estão presentes em todos os computadores, e numa dimensão de centenas ou milhares de computadores, eles podem produzir faltas arbitrárias.

A ideia principal do algoritmo usado no Hadoop BFT é realizar uma votação por maioria para cada tarefa de *map*. Considerando que f é o limite do número de faltas, a estratégia é a seguinte:

- lançar $2f + 1$ réplicas de cada tarefa de *map*
- lançar $2f + 1$ réplicas de cada tarefa de *reduce*; o processamento de cada *reduce* começa quando $f + 1$ cópias dos mesmos dados foram produzidas por diferentes réplica de cada tarefa de *map*; os dados de saída são guardados no HDFS.

Esta estratégia é simples, mas também é ineficiente, porque multiplica o trabalho a ser feito pelo sistema. Por esta razão foram feitos um conjunto de melhoramentos, como a redução a $f + 1$ réplicas, a distribuição de *checksums*, a execução por tentativas e a replicação no HDFS.

3.1.1 Redução a $f + 1$ réplicas

A replicação é o processo de partilha de informação que assegura a consistência dos dados face a recursos redundantes, de forma a melhorar a fiabilidade e a tolerância a faltas. Em vez de se executar apenas uma tarefa, passa-se a executar um conjunto de réplicas. Porém, não é necessário lançar $2f + 1$ réplicas para tolerar f faltas. Ao lançar $f + 1$ réplicas, já é possível verificar se houve f faltas. Neste último caso, ao se detectar f faltas, lança-se f réplicas suplementares para se saber qual é a réplica correcta.

3.1.2 Message Digests

Os *MapOutputs* podem ter o tamanho de *gigabytes*. Para as tarefas de *reduce* não terem que ir buscar $f + 1$, ou $2f + 1$, partições replicadas para efectuar comparações, usa-se *checksums*.

A soma de controlo, ou *checksum*, é um valor calculado a partir de um bloco de dados e é usado para detectar erros durante a transmissão dos dados pela rede, ou durante a escrita dos dados em disco. Para a nova versão, passou-se a usar somas de controlo geradas por funções de resumo criptográfico (funções de *hash* criptográfico).

Uma função de resumo criptográfico é um procedimento determinístico que pega em blocos de dados e retorna uma sequência de bits de tamanho fixo - soma de controlo. Ela assegura a integridade de qualquer mensagem, uma vez que, é rápido gerar um *checksum*, é impraticável gerar uma mensagem a partir do *checksum* e é impraticável encontrar duas mensagens diferentes com o mesmo *checksum*. A função de resumo criptográfico usado na nova plataforma é o SHA-1. A razão de escolha do SHA-1 é porque oferece uma segurança que ainda não foi quebrada e devido à sua popularidade.

Os *checksums* têm uma grande importância no Hadoop MapReduce BFT, porque permitem que o *JobTracker* e as tarefas de *reduce* verifiquem a integridade dos *MapOutputs*, e que concluam que se têm uma maioria de *MapOutputs* iguais apenas comparando os *checksums*.

3.1.3 Distribuição dos *checksums*

O *JobTracker* tem um papel importante na gestão dos *checksums*. Ele recebe os valores gerados pelas tarefas de *map* e distribui-os para as tarefas de *reduce*. A figura 3.1 mostra um exemplo da distribuição dos *checksums*. Na figura existem três *TaskTrackers* que correm em nós diferentes (`TaskTracker{1,2,3}`). Todas as réplicas de *map* processam o mesmo *split*, mas de *DataNodes* diferentes, e todas elas geram o *MapOutput* e o *checksum*. O *MapOutput* é guardado no disco local, e o *checksum* é enviado para o *JobTracker*, tal como se mostra no passo 1 e 2. Após o JobTracker ter recebido uma maioria de *checksums* iguais respeitante à mesma tarefa, ele informa as tarefas de *reduce* que podem iniciar. Esta informação contém o *checksum* da partição correspondente (passo 4 e 5). As

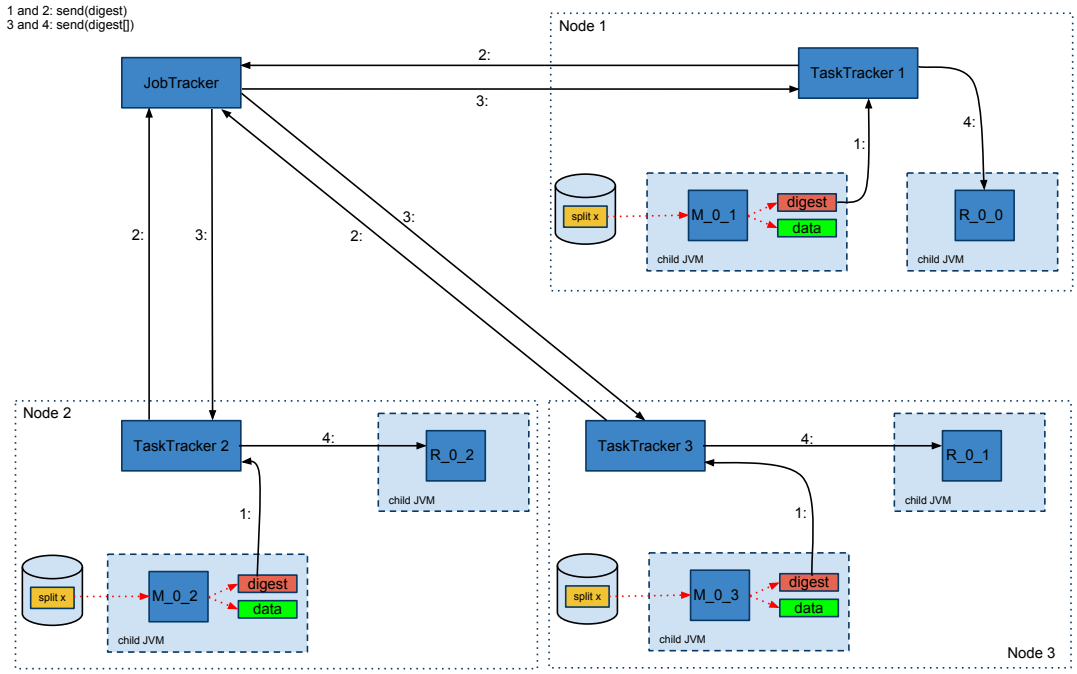

Figura 3.1: Geração e distribuição dos message digests

réplicas de *reduce* vão buscar remotamente a partição correcta, geram um novo *checksum* a partir dos dados e comparam-no com o *checksum* enviado pelo *JobTracker*. Se os *checksums* forem iguais, a réplica de *reduce* continua com a execução. Se não forem iguais, a réplica vai buscar a mesma partição a outro nó.

3.1.4 Execução por tentativas

Ter que esperar que $f + 1$ tarefas de *map* produzam o mesmo *checksum* antes de iniciar as tarefas de *reduce*, pode trazer atrasos à execução do trabalho. Uma melhor forma será lançar executar as tarefas de *reduce* após terem recebido as primeiras cópias dos *MapOutputs*, e enquanto as réplicas de *reduce* estiverem a correr, valida-se os *MapOutputs* à medida que vão aparecendo. Se a um dado ponto é detectado que o *MapOutput* que um *reduce* processou não está correcto, a tarefa é reiniciada.

Esta funcionalidade não foi implementada neste trabalho, mas é indicada como trabalho futuro. Trata-se de uma estratégia que tem como objectivo diminuir o tempo total de execução do trabalho.

3.1.5 Replicação no HDFS

A replicação dos ficheiros de entrada no HDFS, e ao facto de eles estarem associados ao lançamento de uma tarefa *map* por *split*, permite que a computação destas tarefas se faça com dados locais. O *JobTracker* dá preferência em lançar as tarefas de *map* no mesmo

nó que contém o *splits* de entrada. Quando tal nó já possui tarefas demais, outro nó é escolhido dentro do mesmo *rack*, ou em último caso, no mesmo *cluster*.

Na versão BFT, para não se correr o risco de se ter duas réplicas da mesma tarefa de *map* a correr no mesmo nó, modificou-se o factor de replicação do HDFS para ser igual a $2f+1$ e obrigar o *JobTracker* a lançar as réplicas da tarefa em nós distintos e continuarem a obter dados locais.

Respeitante aos dados de saída, por omissão, o HDFS usa um factor de replicação de 3. Na versão BFT, como as tarefas são replicadas, já não existe a necessidade de replicar o resultado final no HDFS, e este parâmetro de configuração passa a ter o valor 1.

Capítulo 4

Implementação

O Hadoop MapReduce BFT é um melhoramento da plataforma oficial, que utiliza a replicação, *checksums* e implementa um sistema de votação simples para que a plataforma seja tolerante a faltas Bizantinas. Implementar no Hadoop a capacidade de ser tolerante a faltas Bizantinas, obriga a fazer um conjunto de alterações, como a instanciação de réplicas, um melhoramento no algoritmo de distribuição de tarefas e criação e distribuição de *checksums*. Este capítulo descreve os detalhes de implementação. As alterações ao código foram feitas a partir da versão 0.20.0, datada de 22 de Abril de 2010.

4.1 Identificação de tarefas

Jobs e tarefas no Hadoop são identificadas por uma cadeia de caracteres, chamados respectivamente de `JobID` e `TaskID`. O primeiro contém o prefixo "`job`", a data de criação do mesmo e um contador (1 based). O segundo contém o prefixo "`task`", seguido da data e do número de contador do *job*, acrescido da letra "m" ou "r", dependendo se é uma tarefa de *map* ou *reduce*, e por um número de sequência. A figura 4.2 apresenta um exemplo de um identificador de trabalho, juntamente com alguns identificadores de tarefas.

- `job_201011231451_0001`
- `task_201011231451_0001_m_000000`
- `task_201011231451_0001_m_000001`
- `task_201011231451_0001_m_000002`
- `task_201011231451_0001_r_000000`
- `task_201011231451_0001_r_000001`

Figura 4.1: Exemplos de identificadores do *job* e de tarefas (Hadoop original)

- `task_201011231451_0001_m_000000_0`
- `task_201011231451_0001_m_000000_1`
- `task_201011231451_0001_m_000000_2`

Figura 4.2: Exemplos de identificadores do *job* e de tarefas (Hadoop BFT)

Ao acrescentar-se réplicas às tarefas, é necessário diferenciá-las uma das outras, e, portanto, duas réplicas de uma mesma tarefa precisam de ser identificadas como tal. Foi acrescentado ao identificador de tarefa um número que identifica qual a réplica da tarefa se refere. Esta forma de identificação permite alterar o Hadoop de forma simples: para cada tarefa criada, passa-se a criar N réplicas da mesma, apenas com identificadores ligeiramente diferentes.

4.2 Instanciação das réplicas

No momento em que as réplicas são instanciadas, em primeiro lugar é criado uma directoria temporária relativa ao *job* que contém os ficheiros de configuração e livrarias. Esta criação do ambiente é feita pela tarefa de setup. A partir daqui é que as tarefas de *map* e de *reduce* são criadas e guardadas numa fila do *job*, mais concretamente, no objecto `JobInProgress`. Esta fila é usada pelo escalonador do *JobTracker* para distribuir o trabalho pelos vários *TaskTrackers*. O contador controla o número de tarefas lançadas.

Listing 4.1: Inicialização de tarefas

```
public synchronized void initTasks()
    throws IOException, KillInterruptedException {

    (...)
    for(int i=0; i < splits.length; ++i) {
  inputLength += splits[i].getDataLength();

  for(int numReplica=0; numReplica<numReplicas; numReplica++) {
      int idx = (i*numReplicas) + numReplica;

      maps[idx] = new TaskInProgress(jobId, jobFile, splits[i],
             jobtracker, conf, this, i, numReplica);

      // set count
      increaseCount(maps[idx].getTIPId().toStringWithoutReplica());
  }
    }

    (...)

  if(numReduceTasks > 0) {
      for (int i = 0; i < numReduceTasks; i++)  {
```

```
    for(int numReplica=0; numReplica<numReplicas; numReplica++) {
        int idx = numReplica + (i * numReplicas);

        reduces[idx] = new TaskInProgress(jobId, jobFile, numMapTasks,
           jobtracker, conf, this, i, numReplica);
        nonRunningReduces.add(reduces[idx]);
    }
      }
  }
    (...)
 }
```

4.3 Input Splits

Um ficheiro pode ser constituído por um ou vários bloco de dados no HDFS. Por omissão, um bloco de dados tem o tamanho de 64MB, no entanto, este valor é configurável. Cada tarefa de *map* processa um único *split*, por isso, o número de tarefas de *map* que um *job* tem, depende do tamanho dos dados de entrada e do tamanho do *split* definido.

Listing 4.2: Quantificação e leitura dos splits

```
public synchronized void initTasks() throws IOException,
    KillInterruptedException {
  (...)
  // read input splits and create a map per a split
  String jobFile = profile.getJobFile();

  Path sysDir = new Path(this.jobtracker.getSystemDir());
  FileSystem fs = sysDir.getFileSystem(conf);

  String jobSplit = conf.get("mapred.job.split.file");// path to job.
     split
  DataInputStream splitFile = fs.open(new Path(jobSplit));

  JobClient.RawSplit[] splits;
  try {
     splits = JobClient.readSplitFile(splitFile);
  } finally {
     splitFile.close();
  }
  (...)
}
```

As tarefas de *map* têm a informação sobre a localização dos *splits*. Se o sistema de ficheiro estiver replicado, cada tarefa de *map* terá um conjunto de localizações diferentes para o mesmo *split*, para se assegurar a disponibilidade dos dados. Na nova plataforma, cada réplica da mesma tarefa passa a ter a mesma lista com as localizações do *split*, contudo cada réplica usará um localização diferente das irmãs. Para se garantir que cada réplica usa o mesmo *split* de *DataNodes* distintos, os ficheiros de entrada têm que estar

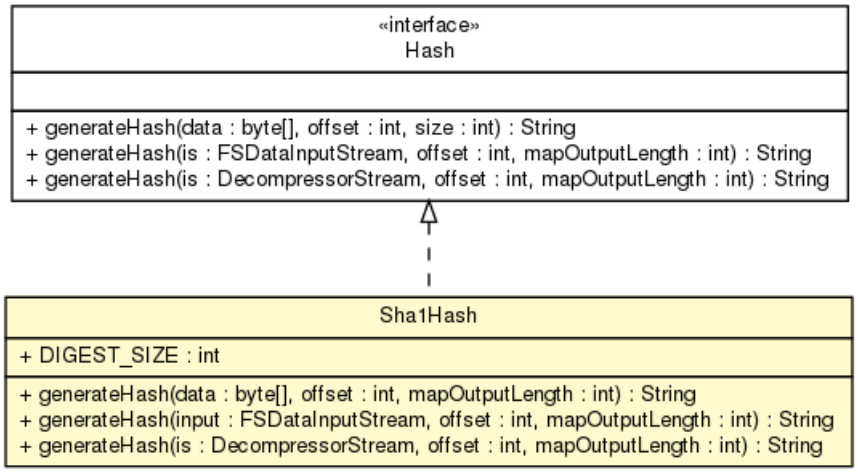

Figura 4.3: Classe e interface para gerar os *checksums* no Hadoop MapReduce BFT

replicados tantas vezes quanto o factor de replicação usado. Ao se usar o mesmo *split* de localizações diferentes e todas as réplicas da mesma tarefa gerarem o mesmo resultado, mais garantias se têm sobre a integridade dos resultados.

4.4 *Checksums*

O uso dos *checksums* tem como objectivo permitir que o *JobTracker* e as tarefas de *reduce* encontrem uma maioria de resultados gerados pelas tarefas de *map*, garantir a integridade dos dados e reduzir a quantidade de dados a transferir.

A interface `Hash` oferece um conjunto de métodos para gerar *checksums*. Na versão BFT optou-se por se usar a função criptográfica SHA-1. A classe `Sha1Hash` implementa os métodos desta interface (figura 4.3).

Os *checksums* recebidos por cada *TaskTracker* é reencaminhado para o *JobTracker* através do *heartbeat*. O *JobTracker* guarda-os até obter uma maioria simples de *checksums* iguais. É através desta contabilização que o *JobTracker* verifica se deve lançar mais tarefas de *map*. Quando o *JobTracker* verificar que tem uma maioria de *checksums* iguais atribuídos à mesma tarefa, ele distribui esta informação para os *TaskTrackers*, para eles poderem reencaminhá-la para as tarefas de *reduce*. O objecto `TaskCompletionEvent` contém a informação necessária sobre a tarefa de *map* para ser usada pelas tarefas de *reduce*.

Os *checksums* também são usados pelas tarefas de *reduce* para validarem os *MapOutput* obtidos remotamente. Após uma tarefa receber o *MapOutput*, ela gerará um novo *checksum* dos dados e irá compará-lo com *checksum* enviado pelo *JobTracker*. Se os *checksums* forem iguais, a tarefa continua, caso contrário, irá fazer uma nova tentativa noutro nó.

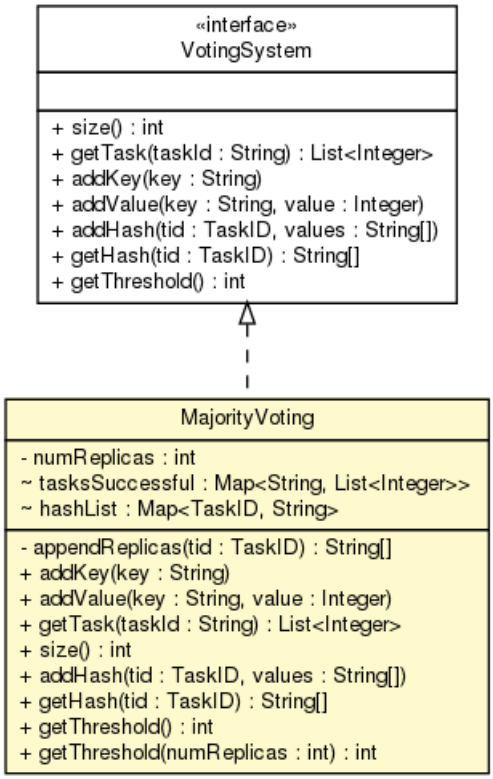

Figura 4.4: Classe e interface usada para guardar os *checksums*

4.5 Redução a $f + 1$ tarefas

Na versão BFT, o esquema básico de execução da plataforma é simples, mas ineficiente, pois implicaria executar todas as réplicas, mesmo nos casos em que algumas delas podiam ser descartadas. Por exemplo, se o *JobTracker* encontra uma maioria num sub-conjunto de réplicas, significa que as restantes réplicas podem ser ignoradas.

Todas as réplicas de *map* lançadas são contabilizadas pelo *JobTracker*, de forma a garantir que sejam lançadas pelo menos $f + 1$ réplicas. Se o *JobTracker* não encontrar $f + 1$ *checksums* iguais pertencentes a uma tarefa, eles lança mais f réplicas.

Todos os *checksums* das tarefas de *map* terminadas são guardados no objecto MajorityVoting (figura 4.4).

4.6 *MapOutput*

Cada tarefa de *reduce* selecciona a réplica de *map* correcta de acordo com o número da réplica do seu identificador, para obter o *MapOutput*. Por exemplo, a réplica de *reduce* 0, da tarefa 0, vai buscar o *MapOutput* à réplica de *map* 0, da tarefa 0. A segunda réplica do *reduce* (replica 1) da mesma tarefa vai buscar os dados à réplica de *map* 1 da mesma tarefa, e assim sucessivamente. Isto é feito porque as réplicas têm o mesmo resultado, e

permite-se paralelizar e equilibrar a carga da obtenção dos dados.

4.7 Parâmetros de configuração

A plataforma contém um conjunto de parâmetros de configuração que estão distribuídos em vários ficheiros. No que diz respeito à plataforma MapReduce, os parâmetros de configuração encontram-se definidos no ficheiro `mapred-site.xml`.

Listing 4.3: Ficheiro de configuração do MapReduce

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<!-- Put site-specific property overrides in this file. -->

<configuration>

<!-- 3 by default //-->
<property>
  <name>tasktracker.tasks.replica</name>
  <value>5</value>
</property>

</configuration>
```

Para a nova plataforma foi criado o parâmetro `tasktracker.tasks.replica`. O novo parâmetro indica o número de réplicas que se quer usar. Por omissão, a plataforma usa o valor 3, e este parâmetro só aceita valores ímpares.

Capítulo 5

Resultados

5.1 Grid'5000

O Grid'5000 é uma infraestrutura distribuída por nove locais em França, usada para o estudo de sistemas paralelos e distribuídos de grande escala. A infraestrutura visa proporcionar uma plataforma altamente configurável e monitorizável pelos utilizadores, com o objectivo de proporcionar um *testbed* que permita que se realizem experiências em todas as camadas de *software*, entre os protocolos de rede até as aplicações. Cada local contém centenas de PCs ligados em reduce, perfazendo um total 1572 nós (ver tabela 5.1) [1].

5.2 GridMix

O Gridmix é uma aplicação que contém um conjunto de testes de benchmarking usados para tirar um conjunto de métricas, como o tempo de execução das tarefas e a quantidade de dados escritos e lidos. Os testes existentes são o MonsterQuery, StreamingSort, JavaSort, WebdataScan, WebdataSort e Combiner, e são descritos nos pontos abaixo.

- WebdataScan - Este teste retorna porções dos ficheiros de entrada. Cada tarefa de *map* cria um par chave-valor a partir do bloco de dados que lê. Os primeiros bytes do bloco é a chave, e o restante é o valor. As tarefas de reduce convertem os pares chave-valor em blocos do dados e o resultado final é guardado em ficheiro no HDFS.

- WebdataSort - Cada tarefa de *map* lê um bloco de dados do ficheiro de entrada e constrói pares chave-valor. Os primeiros bytes é a chave, e o restante é o valor.

Sites	Bordeaux	Grenoble	Lille	Lyon	Nancy	Orsay	Rennes	Sophia	Toulouse	Total
Nodes	154	118	100	135	236	340	162	151	132	1572

Tabela 5.1: Tabela com o número de nós disponíveis

Cada tarefa de reduce ordena os pares chave-valor, concatena os blocos e guarda o resultado final no HDFS.

- StreamingSort - No exemplo do StreamingSort, o *map* e o reduce são processos separados que lêem o input do stdin e enviam o output para o stdout. Cada tarefa de *map* lança o comando UNIX `cat` em processos separados sobre os blocos de dados. As tarefas de *map* pegam no resultado do stdout e convertem-no em pares chave-valor. A primeira sequência de bytes até ao separador é considerado a chave, e o restante é o valor. Cada tarefa de reduce obtém o resultado gerado pelas tarefas de *map*, ordena os pares pela chave e lança o comando `cat` sobre os dados de entrada. O stdout é convertido em blocos de dados e guardado no HDFS. Os pares que não tiverem valor, são descartados pelas tarefas de reduce.

- MonsterQuery - O teste do MonsterQuery é um exemplo que filtra um texto iterativamente, devolvendo apenas porções dele. Cada tarefa de *map* apenas converte 10% do tamanho total ficheiros de entrada em par chave-valor. A primeira palavra é considerada a chave, e o valor é o resto do bloco de dados. Do resultado gerado pelas tarefas de *map*, as tarefa de reduce convertem apenas 40% dos pares chave-valor em blocos de dados e guarda o resultado final no HDFS. Posteriormente, outro job será lançado, que irá ler o resultado gerado da iteração anterior, e fará os mesmos passos descritos atrás. Este processo repete-se mais uma vez. Em suma, a execução de um exemplo do MonsterQuery é constituído pela execução iterativa de 3 jobs.

- JavaSort - O teste do JavaSort tem como objectivo ordenar um texto pela chave. Cada tarefa de *map* converte cada linha dos ficheiros de entrada em pares chave-valor. Por cada linha, a tarefa de *map* vai à procura de uma sequência de bytes para criar pares chave-valor. A parte do texto que estiver atrás da sequência de bytes, é considerada a chave. O que estiver à frente da sequência de bytes é considerado o valor. Se não existir a sequência de bytes, a linha é considerada a chave, e o valor é vazio. Cada tarefa de reduce apenas guarda os valores dos pares criados pelas tarefas de *map* no HDFS.

- Combiner - O teste do Combiner é o exemplo do WordCount que tem definido uma classe de combiner. O conceito combiner no Hadoop MapReduce tem o objectivo de funcionar como uma tarefa de reduce prévia, que recebe os dados de saída de cada *map* e agrega-los antes de os enviarem para as tarefas de reduce. As classes de combiner definidas pelo utilizador são na verdade classes de reduce. O objectivo do combiner é melhorar a eficiência, usando um reduce como uma tarefa intermediária. Os dados gerados pelo combiner serão processados pelas tarefas de reduce, que guardarão o resultado final no HDFS.

Site	Rennes	Lille
Jobs run	MonsterQuery StreamingSort JavaSort	WebdataScan WebdataSort Combiner
Cluster	Paramount	Chicon
Model	Dell PowerEdge 1950	IBM eServer 326m
CPU	Intel Xeon 5148 LV 2.33 Ghz 2 cpus per node 2 cores per cpu	AMD Opteron 285 2.6 GHz / 1 MB / 800 MHz 2 cpus per node 2 cores per cpu
Memory	8 GB	4 GB
Network	Myri-10G Gigabit Ethernet	Myri-10G Gigabit Ethernet
Storage	2x300 GB Raid0 / SATA	80 GB / SATA

Tabela 5.2: Características das máquinas usadas nos testes

O conjunto dos testes foi dividido em dois grupos, e foram conduzidos em 10 nós de dois *sites* diferentes. Um sumário das características do ambiente usado é mostrado na Tabela 5.2. O ambiente escolhido é condicionado pelo número de máquinas, de forma a limitar os recursos utilizados durante os testes.

Os testes MonsterQuery, StreamingSort e JavaSort correram no *site* Rennes, e o testes WebdataScan, WebdataSort e Combiner foram executados no *site* Lille. Foram executados vários exemplos com um número variado de splits. Executaram-se exemplos com 10, 20, 50, 100, 150, 200, 250, 300, 400, 500, 600, 700, 800, 900 e 1000 splits. Como cada tarefa de *map* processa um split, no maior exemplo a ser executado na versão oficial, serão lançadas 1000 tarefas de *map*. Na versão BFT serão lançadas pelo menos 2000 tarefas de *map*.

Foi utilizado um factor de replicação de 3, para se tolerar no máximo 1 falta ($n = 2f + 1$). Considera-se este valor suficiente porque, é mais provável que n réplicas falhem com resultados diferentes, do que retornarem o mesmo resultado falso. Também duas réplicas da mesma tarefa não correm no mesmo nó, diminuindo a probabilidade de devolverem o mesmo resultado final incorrecto, em caso de existirem nós faltosos.

Os parâmetros de configuração dos testes estão especificados no ficheiro `gridmix-config.xml`. Aqui define-se parâmetros como o número de testes, a quantidade de ficheiros de entrada e o número de tarefas de reduce que se quer usar. Foi neste ficheiro que se definiu o número de splits, para o tamanho do problema, e o número de tarefas de reduce que cada teste usou. Foi definido 50 tarefas de reduce para todos os testes.

Todos os resultados dos testes do Gridmix são gravados em ficheiro de texto. Os valores mostrados nos vários gráficos corresponde a uma média de 5 execuções dos dados recolhidos nos ficheiros.

Por omissão, o tamanho do bloco de HDFS é de 64 MB. Este foi o tamanho usado nos testes porque, ao usar-se blocos com o tamanho múltiplo de 64 MB, faria com que os

exemplos se tornassem maiores e se demorasse mais tempo a realizar os testes. O tamanho total de dados usado foi calculado de modo a que cada *split* tenha a mesma dimensão do bloco. Num exemplo que lê 500 splits, é lido um total de 32 GB (500*64MB) de dados. Ao utilizar-se *splits* com um tamanho idêntico ao tamanho do bloco do HDFS, está-se a maximizar o uso das tarefas de *map*.

5.3 Resultados do Gridmix

A primeira análise que se pode fazer para se perceber que atrasos o Hadoop BFT trouxe em relação à versão oficial, é perceber o quanto a plataforma se tornou mais lenta. A figura 5.1 mostra o rácio do tempo de execução total, para se perceber quantas vezes mais a plataforma BFT demora a executar que a versão oficial. O rácio é obtido pela razão entre o tempo de execução total de cada exemplo da versão BFT sobre a versão oficial.

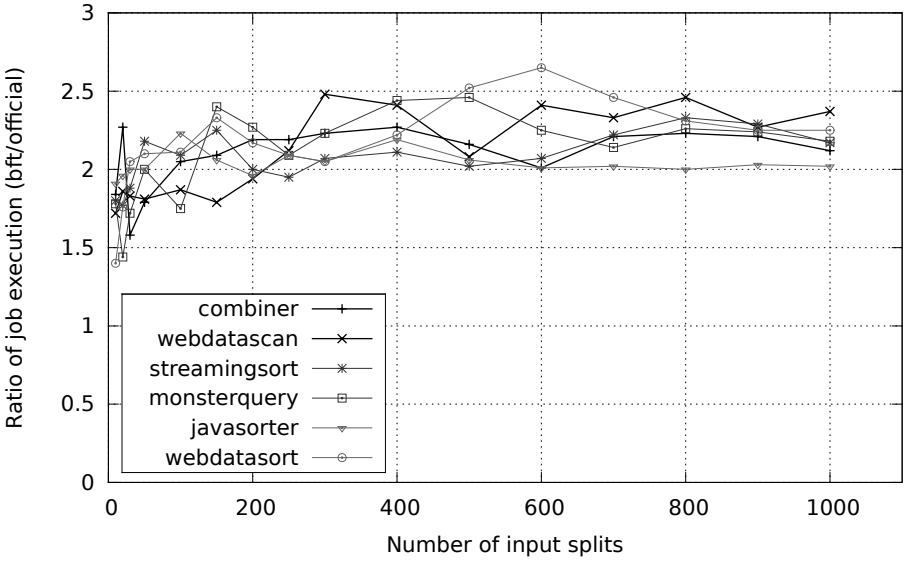

Figura 5.1: Rácio das várias execuções de todos os exemplos do Gridmix

Pode-se observar que a maioria dos exemplos têm um valor de rácio no intervalo de 2 a 2.5. Nos exemplos com poucos *splits* de entrada, este valor tende a ser inferior a 2. A razão do valor ser inferior a 2, é porque à quantidade de dados a processar ainda não é suficientemente grande para introduzir atrasos na execução. Quando se fala em exemplos maiores, como os exemplos em que as tarefas de *map* processam a partir dos 500 *splits*, com esta configuração, na versão BFT, pelo menos 1000 tarefas de *map* irão processar blocos de 64MB e 150 tarefas de *reduce* irão retornar a solução final. Trata-se do dobro de tarefas de *map* e o triplo de tarefas de *reduce* que executaram num conjunto de 10 nós, em tantos nós quanto na plataforma usada para os testes realizados da versão oficial. Devido a se ter aumentado o trabalho e continuar a usar-se a mesma quantidade de recursos, é provável que exista situações em que a quantidade de memória disponível na máquina

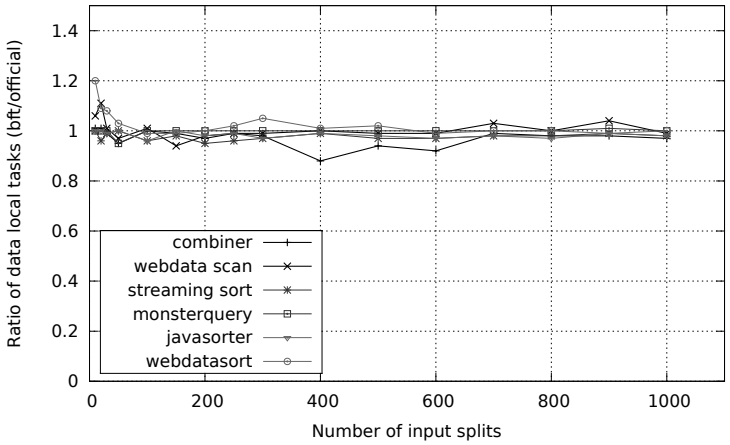

Figura 5.2: Rácio da percentagem de tarefas data-local de todos os exemplos do Gridmix

virtual não seja suficiente, ou as operações de I/O bloqueiem com maior frequência. É nestas situações de bloqueio que é introduzido mais carga nos nós com o lançamento de tarefas especulativas. Uma outra razão para que o desempenho não seja melhor, é porque nestes testes correram $2f + 1$ tarefas de *reduce*, e não $f + 1$. Este ponto vai ser corrigido num trabalho futuro.

Na figura 5.2 mostra-se o rácio da percentagem de tarefas *data-local* (tarefas que correram no mesmo nó aonde se encontra o *split*). Este valor anda em torno de 1, o que mostra que em ambas as versões, a percentagem de tarefas locais lançadas em ambas as versões são semelhantes. Em casos pontuais, a percentagem de tarefas *data-local* da versão BFT é superior à da versão oficial, o que significa que a versão BFT obteve melhores resultados. Tem-se como exemplo o WebdataSort, que em alguns casos mostra um valor de rácio superior a 1. Na situação oposta, tem-se como exemplo o Combiner, que mostra, em algumas situações, a versão oficial obteve melhores resultados. Estas variações de valores dependem da eficiência do algoritmo de distribuição de trabalho.

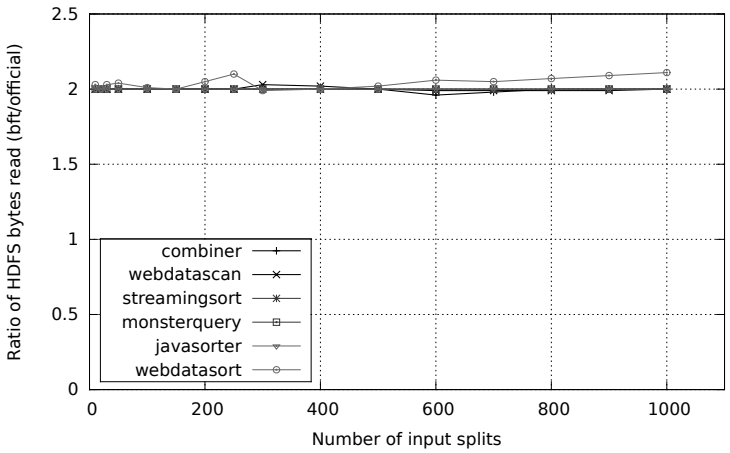

Figura 5.3: Rácio da percentagem de bytes lidos do HDFS

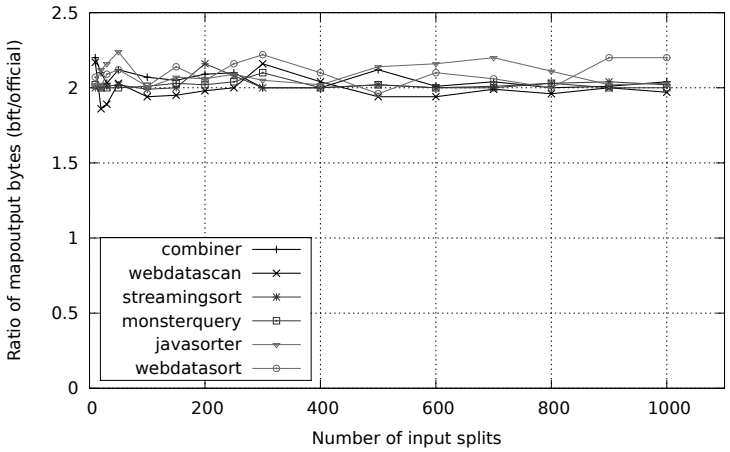

Figura 5.4: Rácio da percentagem de bytes escritos localmente

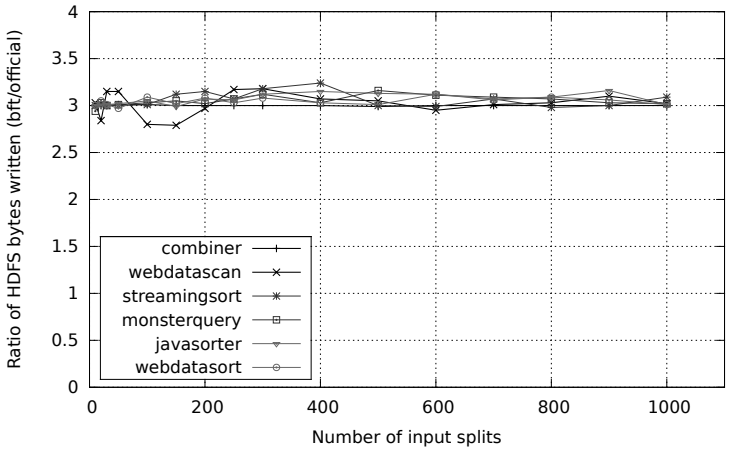

Figura 5.5: Rácio da percentagem de bytes escritos no HDFS

Como são executados $f + 1$ tarefas de *map* na versão BFT, também são gerados mais dados do que na versão oficial. A figura 5.3 mostra que a versão BFT leu duas vezes mais dados de entrada do HDFS, as tarefas de *map* também produziram o dobro de *MapOutput* (ver figura 5.4), e as tarefas de *reduce* produziram 3 vezes mais o resultado final (ver figura 5.5). A contabilização dos dados de saída não inclui a replicação dos dados no HDFS.

Nas secções seguintes mostra-se individualmente os resultados da execução de cada aplicação do GridMix.

5.3.1 WebdataScan

Um conjunto de métricas foram tiradas deste exemplo e são mostradas nos gráficos abaixo.

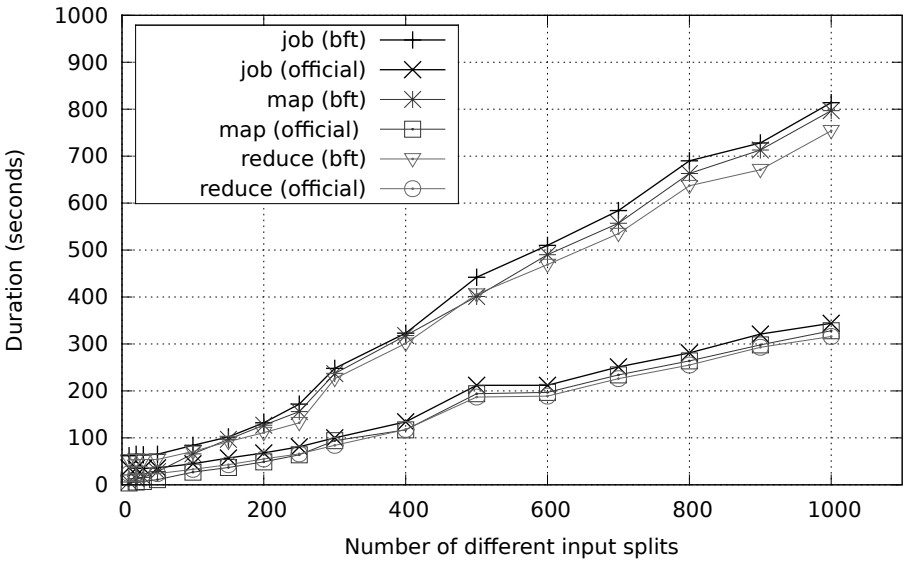

Figura 5.6: Duração do *job*, e das tarefas de *map* e de *reduce*

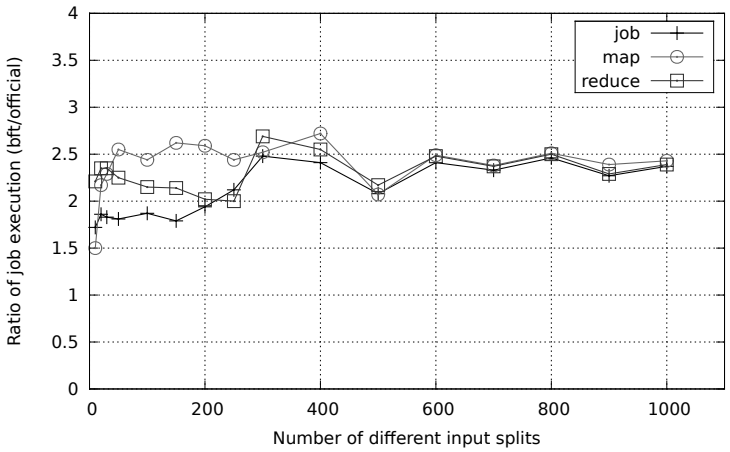

Figura 5.7: Razão entre os tempos totais de execução das duas versões

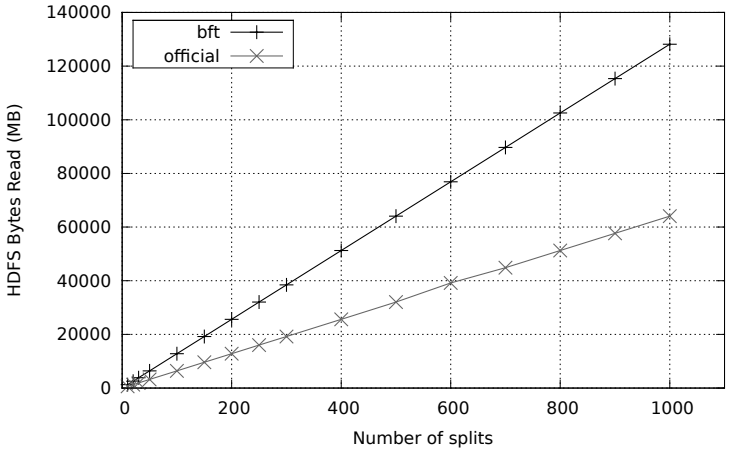

Figura 5.8: Quantidade de dados de entrada lidos

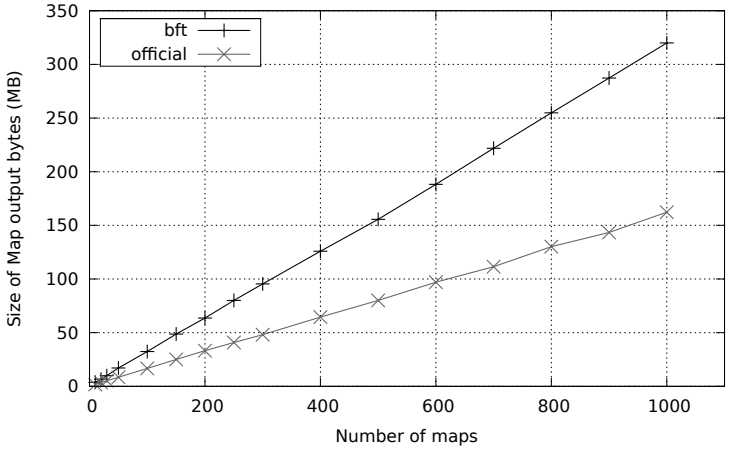

Figura 5.9: Quantidade de *MapOutput* escritos

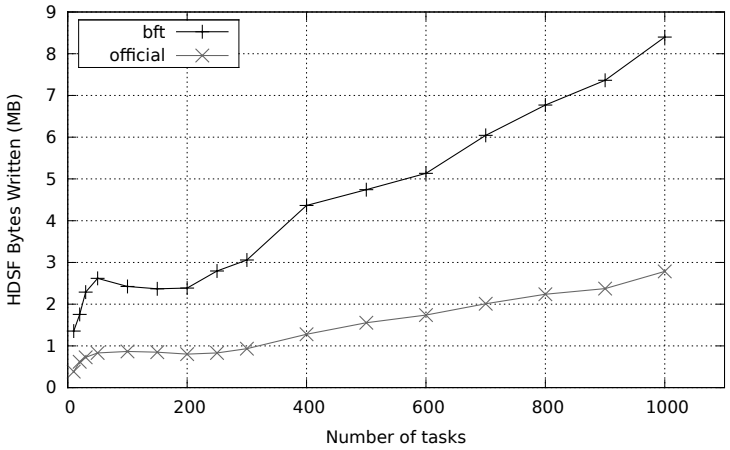

Figura 5.10: Quantidade de dados de saída escritos

Discussão A primeira análise que se pode fazer para se perceber que atrasos o Hadoop BFT trouxe em relação à versão oficial, é perceber o quanto a plataforma se tornou mais lenta. A figura 5.6 mostra o tempo total de execução dos *jobs*. Vê-se na figura que a versão BFT demora sempre mais tempo que a versão oficial. Este comportamento é igual para as tarefas de *map* e de *reduce*. Para se perceber quantas vezes mais a aplicação BFT demorou em relação à oficial, a figura 5.7 mostra o rácio do tempo de execução do *job*. Nos exemplos que correram com poucos *splits*, o rácio do *job* tende a ser inferior 2. A partir dos exemplos que correram com 300, o rácio tende a estabilizar por volta do valor de 2.5. Uma das razões para que esta aplicação demore um valor acima do 2, é devido ao lançamento das tarefas especulativas e pelo facto da aplicação executar $2f + 1$ tarefas de *reduce* em vez de $f+1$. As tarefas especulativas são lançadas pelo *JobTracker* quando elas demoram muito tempo a terminarem. O facto do rácio das tarefas de *map* estar próxima do valor 2.5, mostra que foram lançadas mais tarefas para além de $f + 1$.

Nos exemplos de 300 e 400 *splits*, o rácio do tempo total de execução das tarefas de *reduce* cresceu abruptamente. A fase do *"shuffle & sort"* tem um momento em que a tarefa de *reduce* escolhe se quer guardar todo o *MapOutput* em memória, ou em disco. Nestes exemplos, as tarefas de *reduce* optam por guardar todos estes dados em memória. Esta má decisão rapidamente enche a memória disponível que cada tarefa tem atribuída, fazendo com que demore mais tempo a terminar.

Como são executadas cerca de $f + 1$ mais tarefas de *map* na versão BFT, do que na versão oficial, a quantidade de informação também é duplicada. A figura 5.8 mostra a quantidade de dados de entrada que foram lidos do sistema de ficheiro HDFS. Na versão oficial foram lidos no máximo cerca de 64GB de dados, enquanto que, na versão BFT foram lidos o dobro. Estes dados são processados pelas várias tarefas de *map*. Se se dividir o valor total de dados lidos sobre o número de tarefas de *map*, observa-se que em ambas as versões, cada uma das tarefas de *map* processou 64MB de dados. Num exemplo que correu 100 tarefas de *map*, na versão oficial foram lidos 100 *splits* de 64MB cada, enquanto que na versão BFT foram lidos pelo menos 200 *splits* do mesmo tamanho.

As tarefas de *map* geraram um total de *MapOutput* muito menor ao número de bytes lidos do HDFS. A figura 5.9 mostra que foram gerados no máximo 320 MB de dados. Neste exemplo, cada tarefa de *map* gerou um *MapOutput* com 0.25% do tamanho dos dados lidos, o que faz com que se gaste menos tempo na fase de "*Shuffle*". As tarefas de *reduce* transformam parte dos pares chave-valor em blocos de dados.

No final da execução dos testes foi criado no máximo um total de 2.7 MB na versão oficial e 8 MB na versão BFT.

5.3.2 WebdataSort

Um conjunto de métricas foram tiradas deste exemplo e são mostradas nos gráficos abaixo.

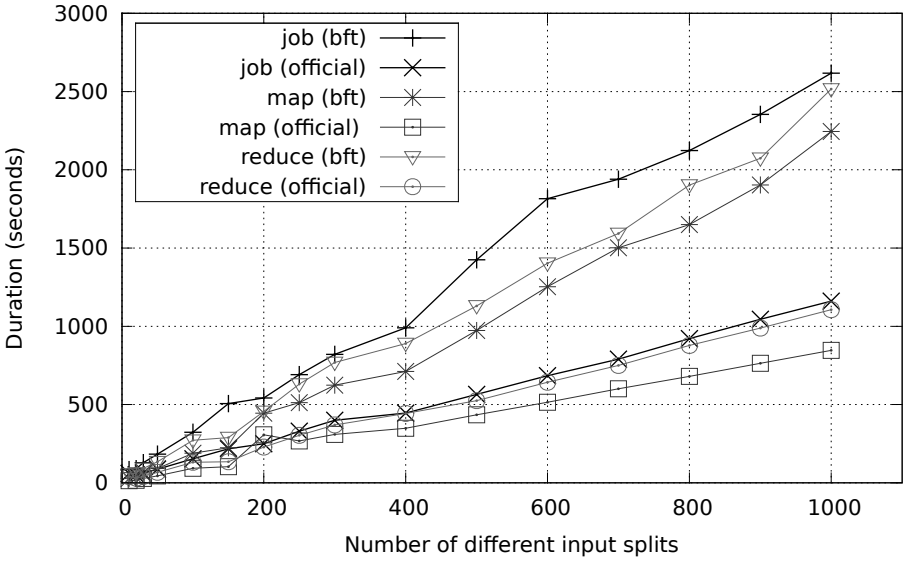

Figura 5.11: Duração do *job*, *map* e *reduce*

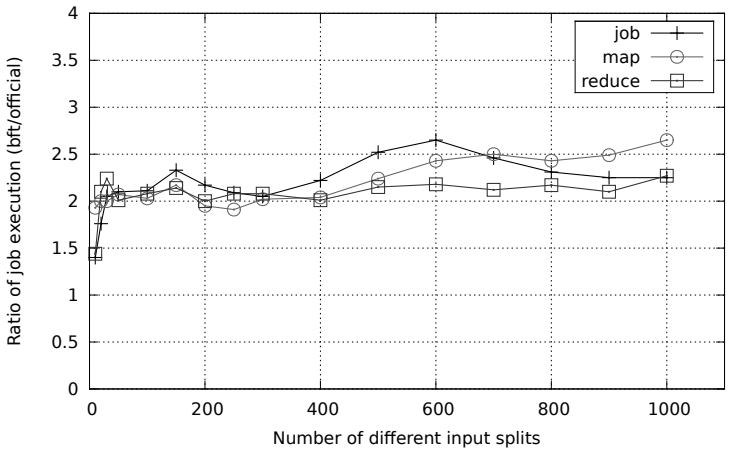

Figura 5.12: Razão entre os tempos totais de execução das duas versões

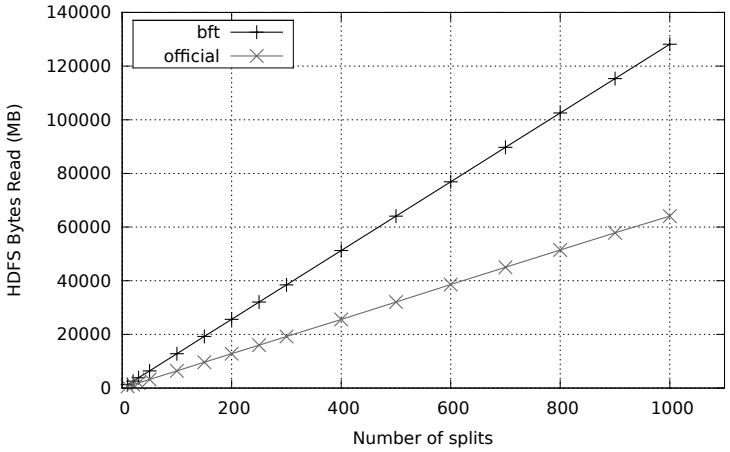

Figura 5.13: Quantidade de dados de entrada lidos

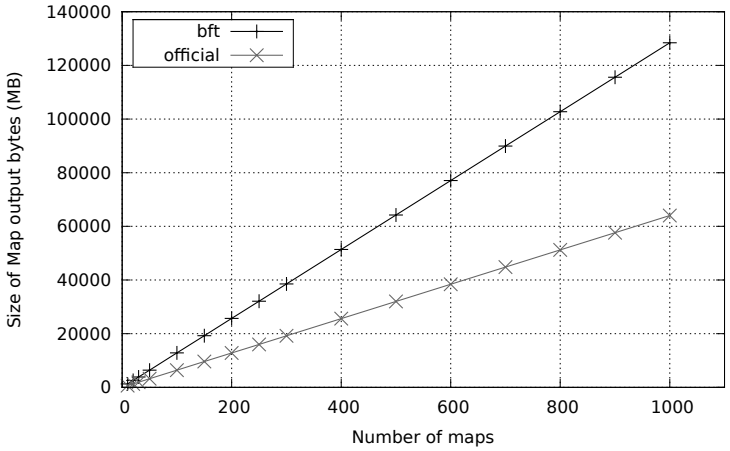

Figura 5.14: Quantidade de *MapOutput* escritos

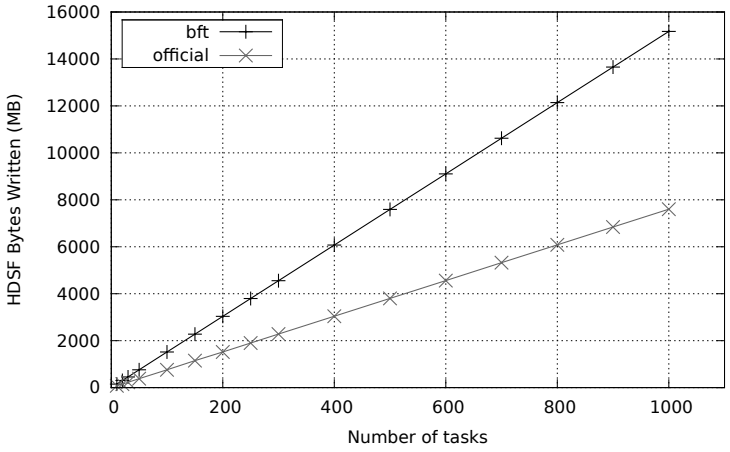

Figura 5.15: Quantidade de dados de saída escritos

Discussão Este exemplo demora mais tempo a executar que o exemplo anterior em ambas as versões. A figura 5.11 mostra o tempo total de execução dos *jobs*. Mas a nível de rácio, observa-se que este exemplo apresenta valores mais regulares. Vê-se na figura que o crescimento da linha BFT é maior que o da versão oficial. Vê-se na figura 5.12 que o rácio do tempo de execução do *job* anda em torno de intervalo 2 e 2.5.

A razão do rácio do *job* estar próximo do valor 2.5 nos exemplos entre os 500 e 700 *splits*, deveu-se à lentidão do *JobTracker* considerar o trabalho terminado. Por uma razão desconhecida, o *JobTracker* demorou mais tempo a receber informação dos *TaskTrackers* sobre o fim das tarefas. Mas curiosamente, o valor do rácio das tarefas de *reduce* é estável, ao contrário das tarefas de *map* que apresentam uma ligeira subida.

A figura 5.13 apresenta os mesmos valores que no exemplo anterior. Na versão oficial, o maior teste leu cerca de 64 GB de dados de entrada, enquanto que, na versão BFT foram lidos o dobro. Foram gerados a mesma quantidade de *MapOutput*, quanto os bytes lidos (ver figura 5.14). No entanto, foram escritos no máximo 7.6 GB de dados no HDFS na versão oficial, e 15 GB de dados para a versão BFT.

5.3.3 Combiner

Um conjunto de métricas foram tiradas deste exemplo e são mostradas nos gráficos abaixo.

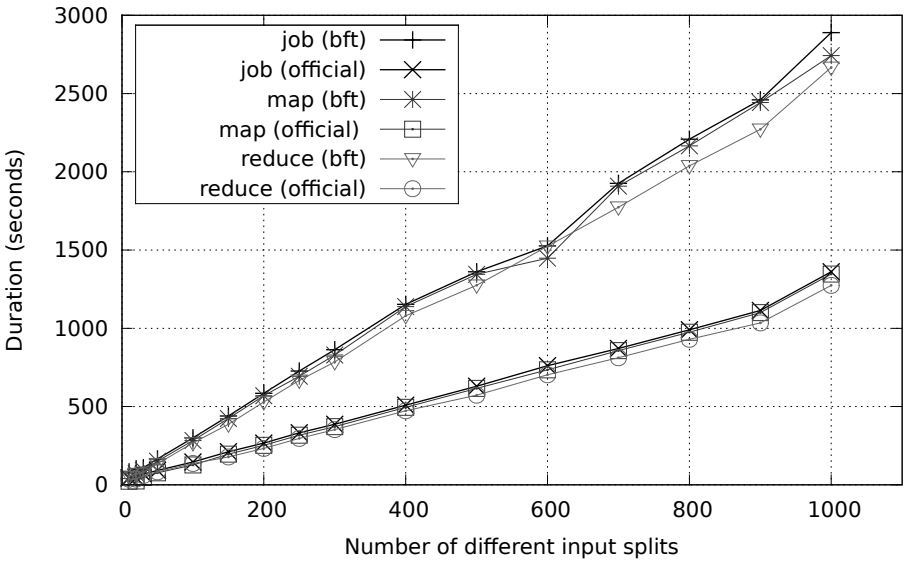

Figura 5.16: Duração do *job*, *map* e *reduce*

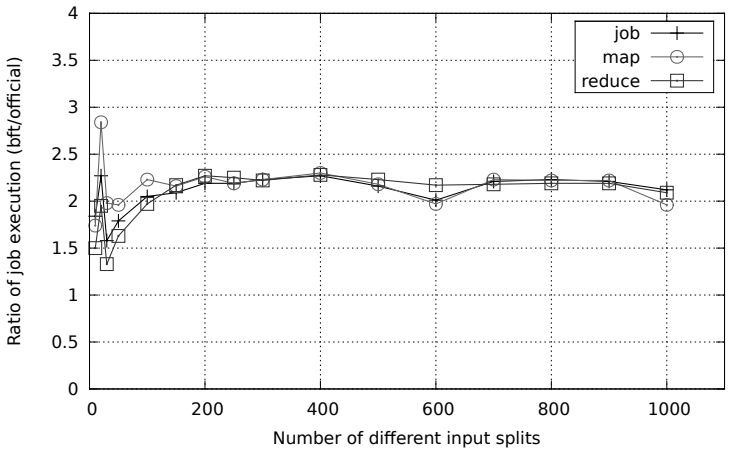

Figura 5.17: Razão entre os tempos totais de execução das duas plataformas

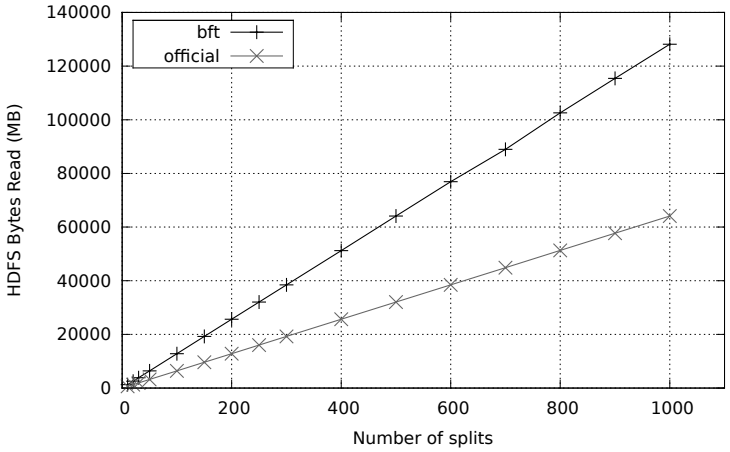

Figura 5.18: Quantidade de dados de entrada lidos

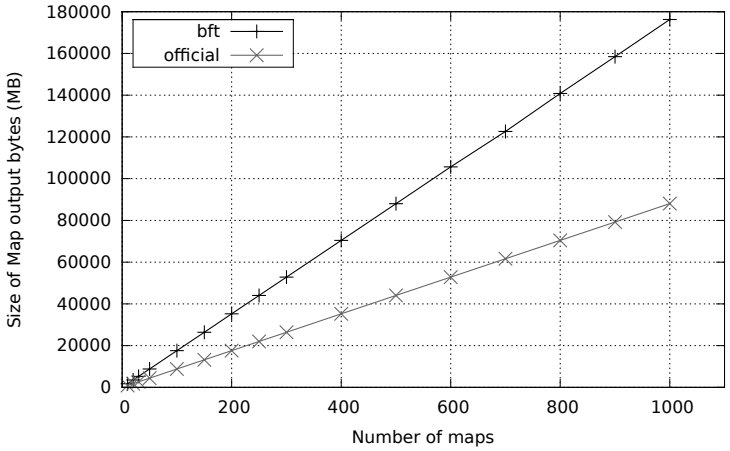

Figura 5.19: Quantidade de *MapOutput* escritos

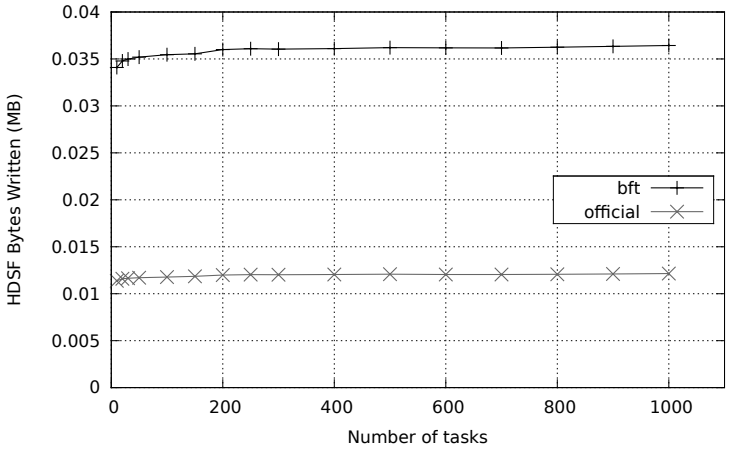

Figura 5.20: Quantidade de dados de saída escritos

Discussão A primeira análise que se pode fazer para se perceber o quanto a versão BFT está mais lenta em relação à versão oficial. A figura 5.16 mostra o tempo total de execução dos vários *jobs*, com um número variado de *splits*. O crescimento da linha BFT acompanha a da versão oficial. Este comportamento é acompanhado pelas tarefas de *map* e de *reduce*. A figura 5.17 mostra que o *job* ao longo dos testes demora cerca de duas vezes mais que a versão oficial.

A figura 5.18 mostra a quantidade de dados que foram lidos do sistema de ficheiro HDFS. Na versão oficial foram lidos cerca de 64 GB de dados do sistema de ficheiro HDFS, enquanto que, na versão BFT foram lidos o dobro. No exemplo com um número maior de *splits*, do total de dados de entrada lidos, foram gerados no máximo *MapOutputs* com um total de 88 MB para a versão oficial, e o dobro para a versão BFT. Por sua vez, as tarefas de *reduce* produziram 12 KB de dados de saída na versão oficial, e o triplo na versão BFT (ver figura 5.20). Embora os resultado pareçam no gráfico constantes, eles variam desde os 11.37 KB até 12.14 KB na versão oficial e, desde os 34.08 KB até 36.42 KB para a versão BFT. Os dados de saída são compostos por pares "palavra-número de ocorrências", daí não haver grande variação no tamanho total dos dados produzidos em ambas as versões.

5.3.4 StreamingSort

Um conjunto de métricas foram tiradas deste exemplo e são mostradas nos gráficos abaixo.

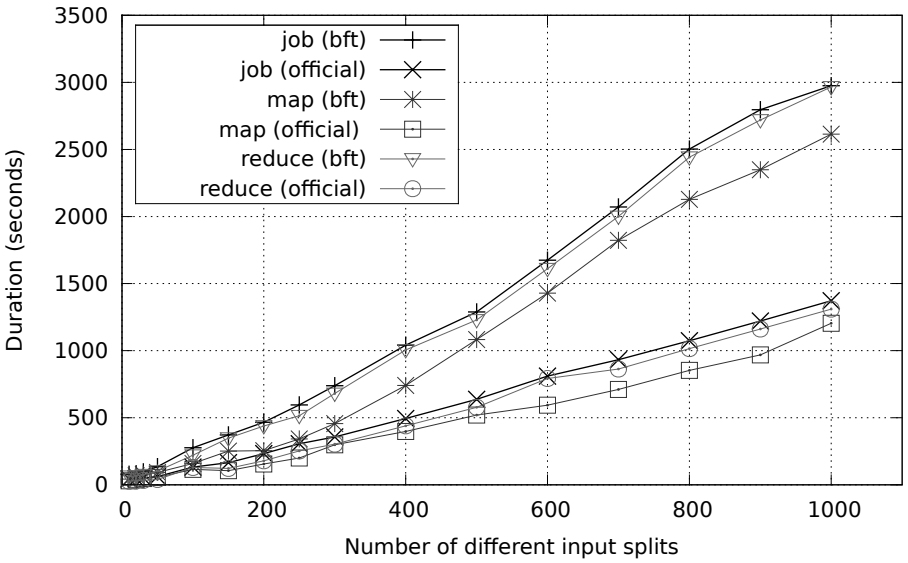

Figura 5.21: Duração do *job*, *map* e *reduce*

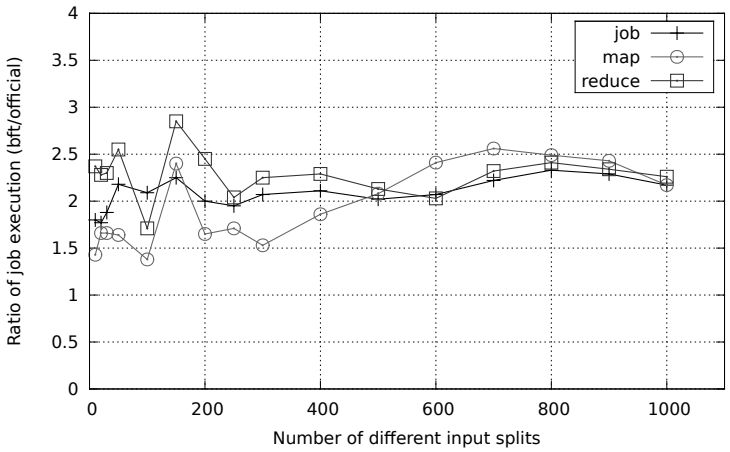

Figura 5.22: Razão entre os tempos totais de execução das duas versões

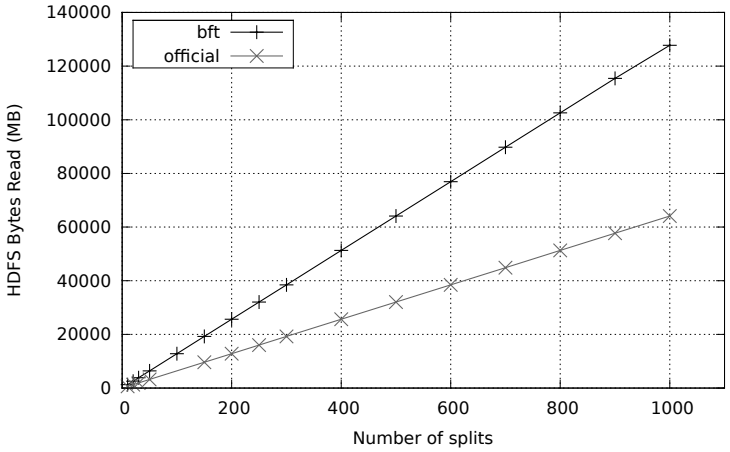

Figura 5.23: Quantidade de dados de entrada lidos

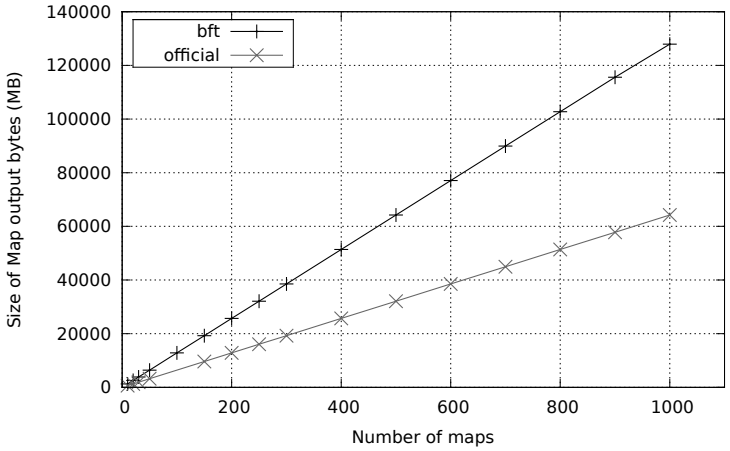

Figura 5.24: Quantidade de *MapOutput* escritos

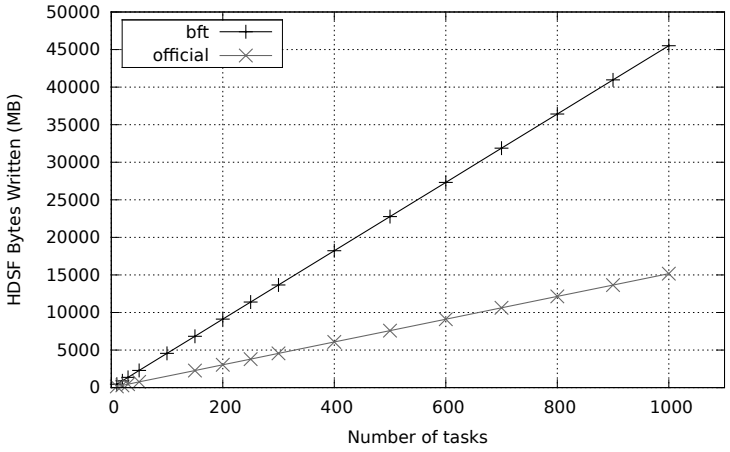

Figura 5.25: Quantidade de dados de saída escritos

Discussão Vê-se na figura 5.21 que a versão BFT demora sempre mais tempo que a versão oficial. No entanto, o rácio das tarefas de *reduce* apresentam flutuações nos exemplos até ao 200 splits. Nos exemplos maiores, o rácio das tarefas tende a estabilizar. É provável que a razão destas flutuações se deva à falta de memória disponível na máquina virtual. No entanto, o rácio do *job* tende a apresentar valores mais constantes.

Os gráficos 5.23, 5.24 e 5.25 apresentam valores idênticos aos exemplos anteriores. No final da execução, no maior exemplo, foi gravado no HDFS 15 GB na versão oficial e 45 GB na versão BFT.

5.3.5 MonsterQuery

Um conjunto de métricas foram tiradas deste exemplo e são mostradas nos gráficos abaixo.

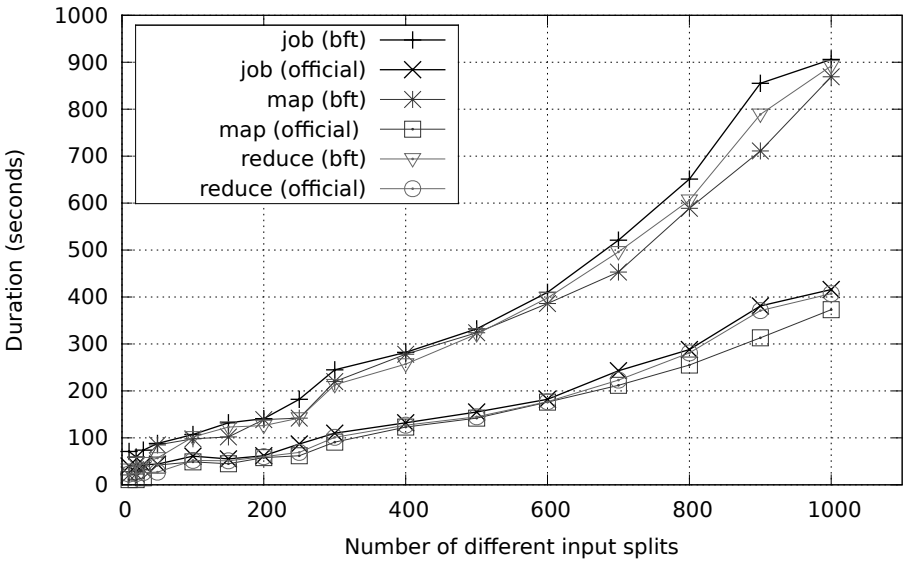

Figura 5.26: Duração do *job*, *map* e *reduce*

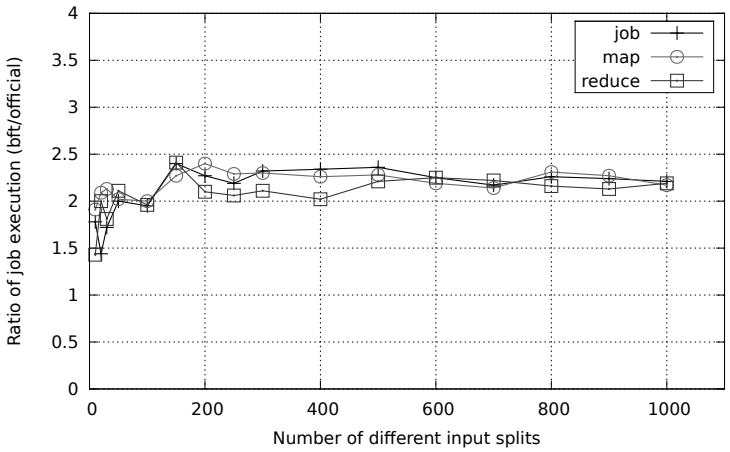

Figura 5.27: Razão entre os tempos totais de execução das duas versões

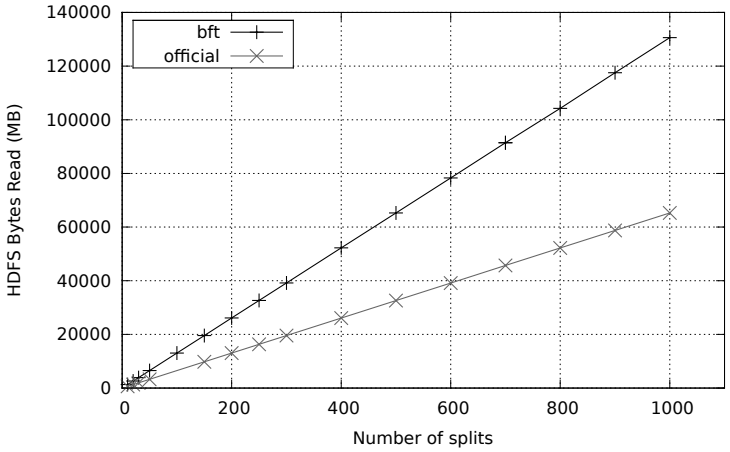

Figura 5.28: Quantidade de dados de entrada lidos

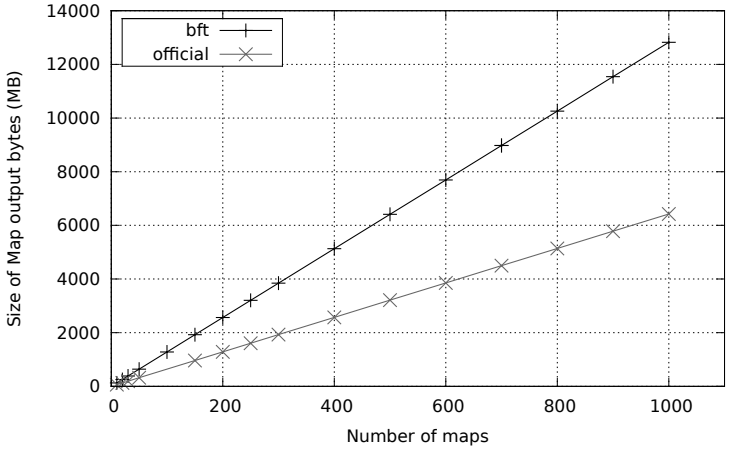

Figura 5.29: Quantidade de *MapOutput* escritos

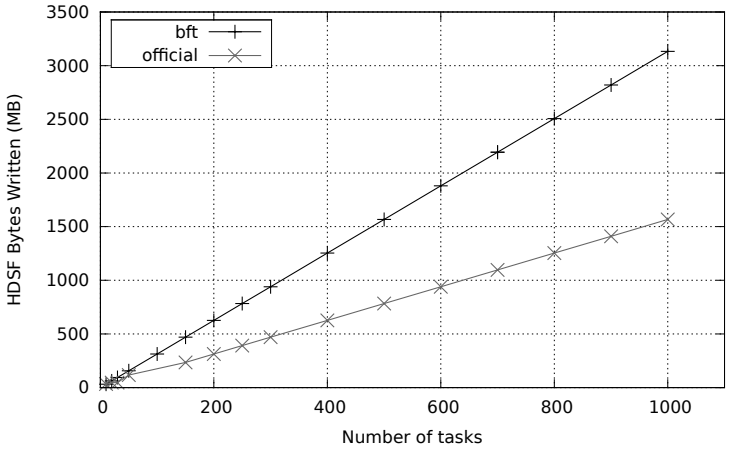

Figura 5.30: Quantidade de dados de saída escritos

Discussão A primeira análise que se pode fazer para se perceber que atrasos o Hadoop BFT trouxe em relação à versão oficial, é perceber o quanto a plataforma se tornou mais lenta. A figura 5.26 mostra o tempo total de execução dos *jobs*. Vê-se na figura que a versão BFT demora sempre mais tempo que a versão oficial. Este comportamento é igual para as tarefas de *map* e de *reduce*. Para se perceber quantas vezes mais a aplicação BFT demorou em relação à oficial, a figura 5.27 mostra o rácio do tempo de execução do *job*. O rácio tende a ficar entre os 2 e 2.5 vezes mais. Uma das razões para que esta aplicação demore um valor acima do 2, é devido ao lançamento das tarefas especulativas, especialmente nos exemplos entre 150 e 250 *splits*. Nestes exemplos, vê-se que o rácio das tarefas de *map* sobe ligeiramente para os valor 2.5.

Os gráficos 5.28, 5.29 e 5.30 apresentam valores idênticos aos exemplos anteriores. Do total de dados gerados pelo *MapOutput*, apenas 25% foi convertido em blocos de dados pelas tarefas de *reduce*. A figura 5.30 mostra que no maior exemplo foram gravados em HDFS 1.5 GB de dados na versão oficial e 3 GB na versão BFT.

5.3.6 JavaSort

Um conjunto de métricas foram tiradas deste exemplo e são mostradas nos gráficos abaixo.

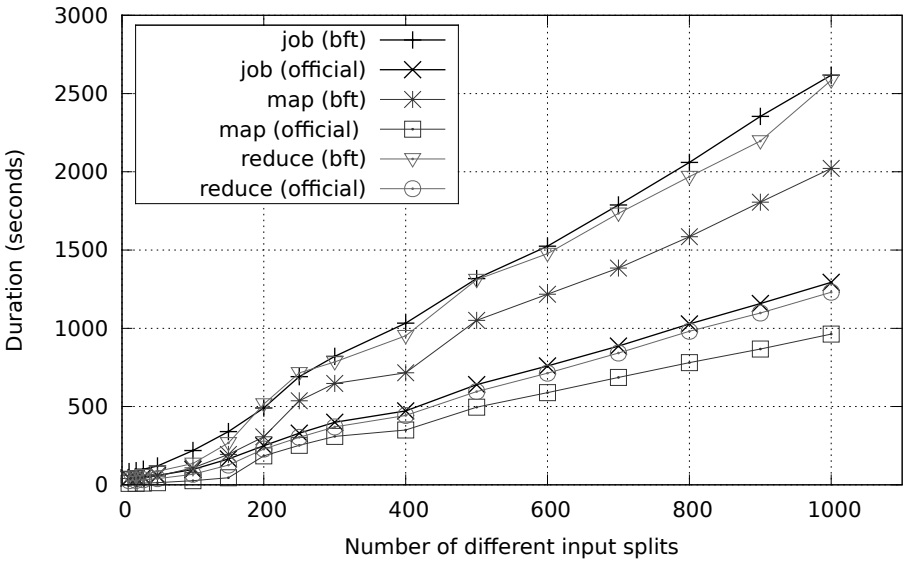

Figura 5.31: Duração do *job*, *map* e *reduce*

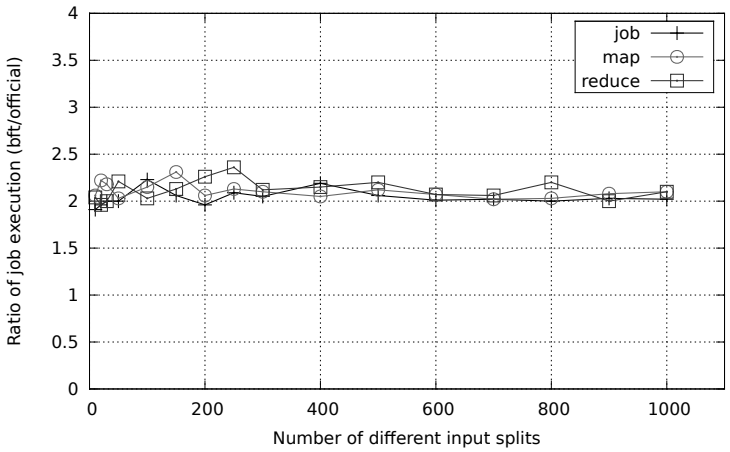

Figura 5.32: Razão entre os tempos totais de execução das duas versões

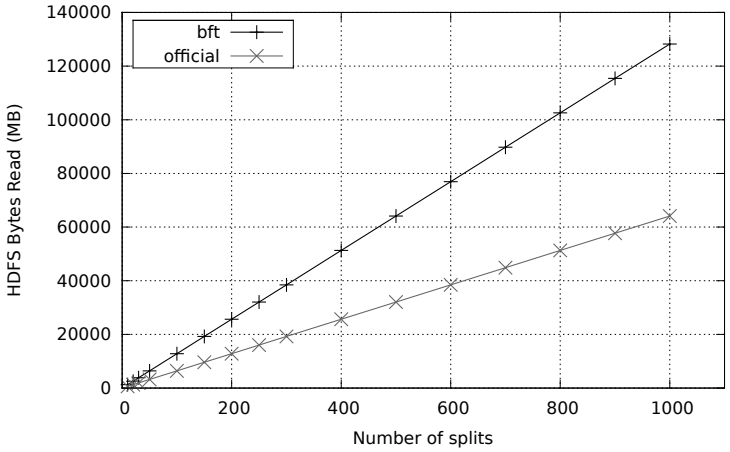

Figura 5.33: Quantidade de dados de entrada lidos

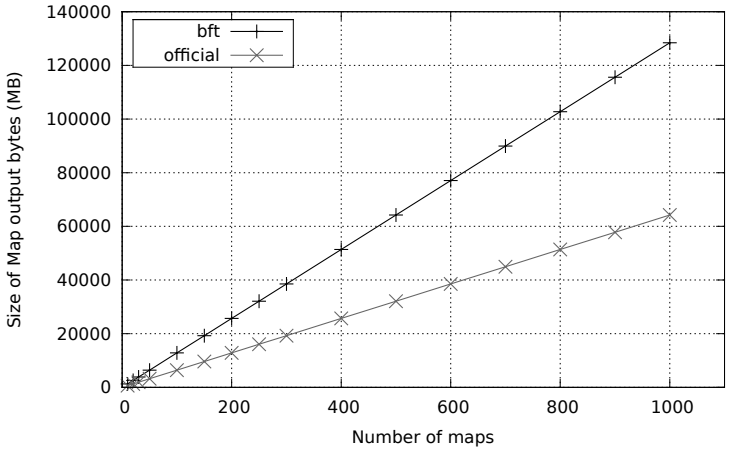

Figura 5.34: Quantidade de *MapOutput* escritos

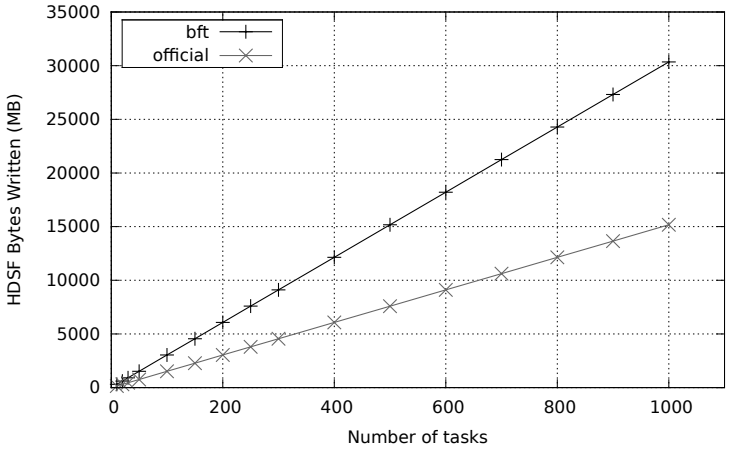

Figura 5.35: Quantidade de dados de saída escritos

Discussão A primeira análise que se pode fazer para se perceber que atrasos o Hadoop BFT trouxe em relação à versão oficial, é perceber o quanto a plataforma se torna mais lenta. A figura 5.31 mostra o tempo total de execução dos *jobs*. Vê-se na figura que o tempo de execução total das tarefas de *map* é bastante menor que as tarefas de *reduce*. Em ambas as versões, as tarefas de *map* demoraram mais tempo a executar que as tarefas de *reduce*.

A figura 5.32 mostra que a versão BFT apresenta um rácio com um valor estável ao longo dos vários exemplos. Os vários exemplos demoram cerca de duas vezes mais que a versão oficial. Este valor sofre pequenas variações ao longo dos vários exemplos, mas nada de significativo.

Os gráficos 5.33, 5.34 e 5.35 apresentam valores idênticos aos exemplos anteriores. Neste exemplo, no máximo foram escritos no HDFS 15 GB na versão oficial e 30 GB na versão BFT.

5.3.7 Avaliação com mais recursos disponíveis.

Os testes realizados do Gridmix mostram que com os mesmos recursos, a versão BFT tende a demorar entre duas a três vezes mais que a versão oficial. A questão de testar um dos exemplos com o dobro dos recursos torna-se inevitável, para perceber-se se a versão BFT consegue apresentar resultados semelhantes à versão oficial. Por esta razão, decidiu-se em pegar num dos exemplos do Gridmix e corrê-lo com o dobro de recursos. O exemplo escolhido aleatoriamente foi o WebdataScan. O exemplo na versão BFT correu em 20 máquinas, e o da versão oficial correu em 10 máquinas.

Um conjunto de métricas foram tiradas deste exemplo e são mostradas nos gráficos abaixo.

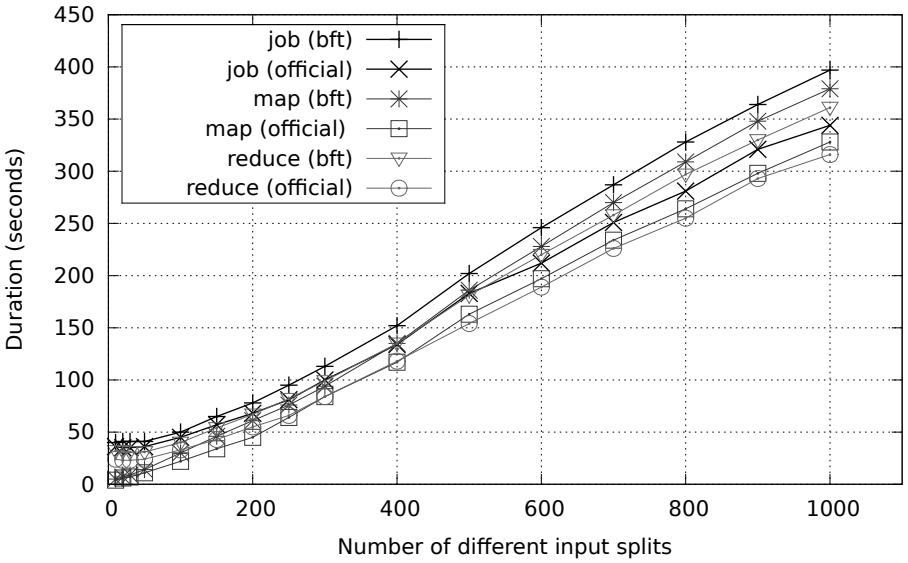

Figura 5.36: Duração do *job*, e das tarefas de *map* e de *reduce*

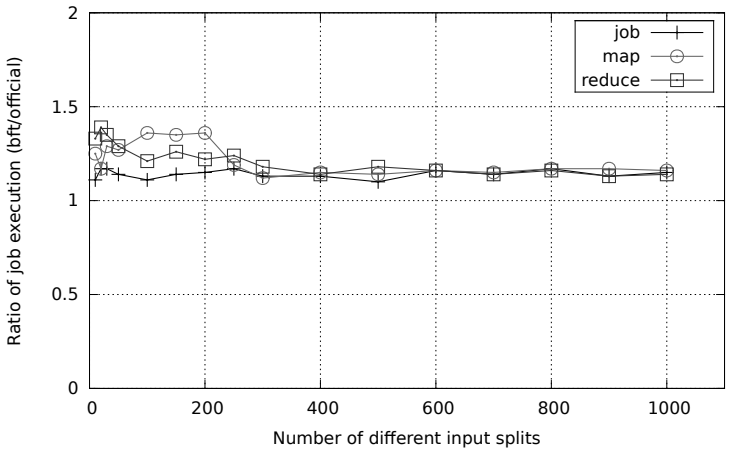

Figura 5.37: Razão entre os tempos totais de execução das duas versões

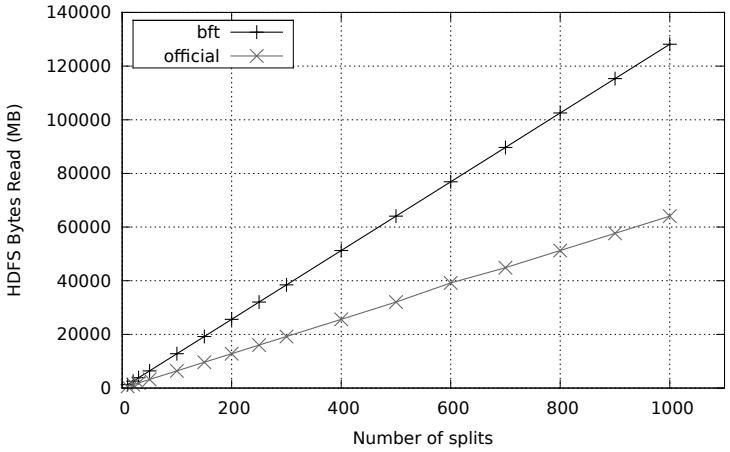

Figura 5.38: Quantidade de dados de entrada lidos

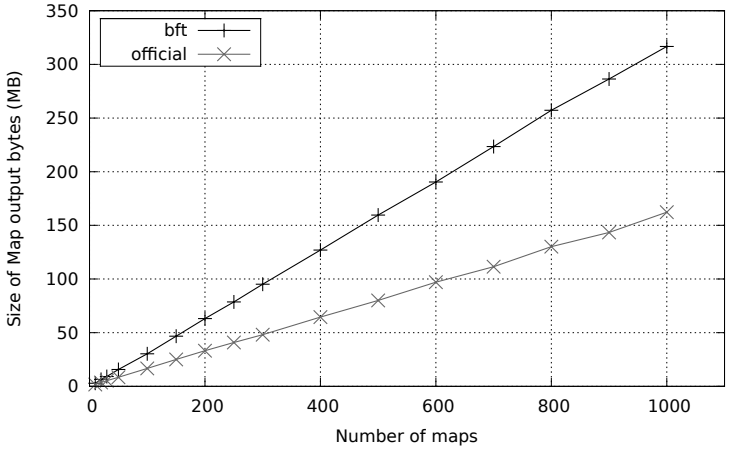

Figura 5.39: Quantidade de *MapOutput* escritos

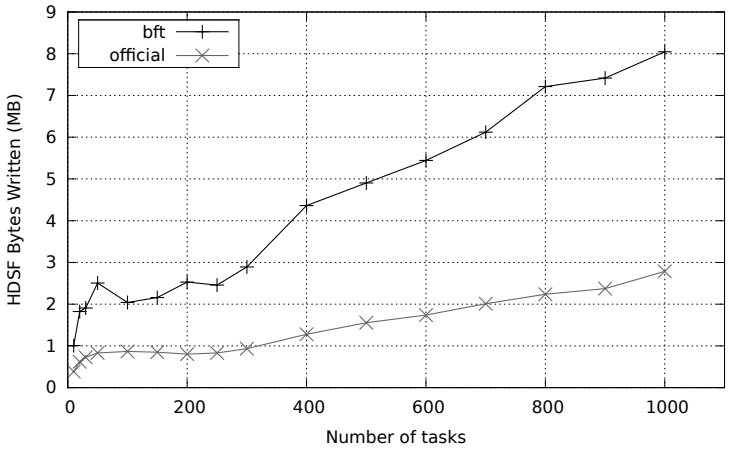

Figura 5.40: Quantidade de dados de saída escritos

Discussão A primeira análise que se pode fazer para se perceber que atrasos o Hadoop BFT trouxe em relação à versão oficial, é perceber o quanto a plataforma se tornou mais lenta. A figura 5.36 mostra o tempo total de execução dos *jobs*, das tarefas de *map* e de *reduce* em ambas as versões. Vê-se na figura que a versão BFT, com o dobro de nós, tem tendência a demorar o mesmo tempo que a versão oficial. Este comportamento é igual para as tarefas de *map* e de *reduce*. Para se perceber quantas vezes mais a aplicação BFT demorou em relação à oficial, a figura 5.37 mostra o rácio do tempo de execução do *job*. Nos vários exemplos, a versão BFT tem tendência a demorar 1.13 vezes mais que a versão oficial. É com muita probabilidade que o facto do valor ser superior a 1, deve-se ao facto de a aplicação executar $2f + 1$ tarefas de *reduce* em vez de $f + 1$. De qualquer forma, a versão BFT, com o dobro de recursos, tem tendência a apresentar resultados parecidos com a versão oficial. Pode-se prever que, se apenas se lançasse $f + 1$ tarefas de *reduce*, o desempenho da versão BFT seria melhor.

Os gráficos 5.38, 5.39 e 5.40 apresentam valores parecidos com os do teste mostrado na secção 5.3.1. No final da execução dos testes foi criado no máximo um total de 2.7 MB na versão oficial e 8 MB na versão BFT.

Capítulo 6

Trabalho Futuro

Este trabalho é um melhoramento da plataforma oficial, em que utiliza as técnicas replicação e um sistema de votação simples para mascarar faltas. No entanto, existem outros pontos que podem ser implementados para melhorar o desempenho da plataforma, e existem algumas vulnerabilidades, que foram adquiridas da versão oficial, que podem pôr em causa a integridade da plataforma através de ataques maliciosos. Este trabalho não visa cobrir ataques maliciosos, mas é listado à mesma possíveis problemas que a aplicação possa ter neste âmbito.

O Hadoop MapReduce lança todas as tarefas de *reduce*, mas poderia-se lançar apenas um $\frac{1}{3}$ delas. Ao se realizar esta melhoria, menos tarefas de *reduce* seriam lançadas, o que aumentaria o desempenho da plataforma. Juntamente com esta funcionalidade, teria que existir um mecanismo de validação do resultado final gerado pelas tarefas de *reduce*, para que o *JobTracker* saiba se é necessário lançar mais tarefas de *reduce*, ou se o *job* terminou com sucesso.

A funcionalidade que permitia a execução de tarefas por tentativa não foi implementada - *Tentative Execution* (ver secção 3.1.4). Esta funcionalidade obrigaria uma alteração profunda no Hadoop MapReduce, o que era impossível de se realizar por falta de tempo, deixando este ponto para trabalho futuro.

Se todas as tarefas de *reduce* validassem o resultado final antes do escrevem no HDFS, apenas seria necessário uma réplica de uma tarefa de *reduce* guardar os dados, e não todas. Ao diminuir o número de escritas no HDFS no lado *reduce*, especialmente nos casos em que é necessário escrever Gigabytes ou Terabytes de dados, o desempenho da plataforma melhoraria.

Os componentes da plataforma comunicam entre si por chamadas RPC e por HTTP. Os dados que são enviados para os vários componentes vão em claro (*"plain text"*) e não oferecem nenhuma garantia sobre a autenticidade do emissor. Qualquer *"hacker"* pode comprometer a confidencialidade (*"eavesdropping"*), ou simular a sua identidade (*"man-in-the-middle attack"*) para manipular a informação. Uma solução para este problema consiste em garantir a autenticidade dos vários intervenientes (componentes) e todas as

mensagens passarem a ser assinadas e cifradas.

O *JobTracker* e o *NameNode* continuam a ser *"single point of failure"*. Se um desses componentes ficar comprometido, a plataforma deixa de funcionar correctamente. Uma solução seria introduzir a capacidade destes componentes reiniciarem-se, ou instalá-los em sistemas tolerante a intrusões, ou ter mais do que uma instância. Se se implementasse o último caso, se um destes componentes falhasse, as instâncias de *backup* passariam a lidar com o resto da execução.

Todo o código criado dentro das funções de `map` e de `reduce` é da responsabilidade do utilizador, e não é controlado pela plataforma. A partir destas funções, o utilizador pode aceder ao sistema operativo, ao sistema de ficheiros, às directorias temporárias, ou à rede. É necessário controlar acessos indevidos, usando o Serviço de Autenticação e Autorização do Java (JAAS), para evitar qualquer tipo de ataque malicioso.

Capítulo 7

Conclusão

Este trabalho apresenta um algoritmo MapReduce tolerante a faltas Bizantinas e o respectivo protótipo. O algoritmo utiliza $f + 1$ réplicas para tolerar faltas arbitrárias, apesar da maioria dos algoritmos de BFT necessitar de $3f + 1$ réplicas. O protótipo foi extendido sobre a plataforma Hadoop MapReduce, o que obrigou a alteração de algumas partes do código fonte.

A nova versão foi testada com aplicações reais provenientes da ferramenta de *benchmarking* Gridmix na infraestrutura Grid'5000. Com estes testes mostra-se que o protótipo tolera faltas arbitrárias sem diminuir o desempenho mais do que o necessário. Os resultados dos testes confirmam o que poderia ser intuído a partir do algoritmo: que com $f = 1$, o tempo para executar um trabalho num pequeno *cluster* duplica, e não triplica, o que equivale a aproximadamente o dobro do tempo de CPU. O uso de $f = 1$ nos testes é uma assunção realista porque: (i) as faltas arbitrárias são raras; (ii) isso significa que a probabilidade de mais do que uma réplica da mesma tarefa retornar o mesmo resultado errado é insignificante.

Como com o mesmo número de recursos usados nos testes, os exemplos corridos na versão BFT demoram cerca do dobro do tempo que na versão oficial. Também foram apresentadas algumas funcionalidades em trabalho futuro para melhorar o protótipo.

O trabalho desenvolvido nesta tese tornou possível várias publicações. Dois resumos foram publicados em conjunto com a equipa do laboratório Francês LIP6, apresentados na conferência "40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2010)" [6] e na conferência "Rencontres francophones du Parallélisme (RenPar'20)" [11]. Um artigo completo descrevendo o trabalho da tese foi publicado na conferência "IEEE CloudCom 2011" [11].

Bibliografia

[1] Hardware of Grid'5000. `https://www.grid5000.fr/mediawiki/index.php/Special:G5KHardware`.

[2] Luciana Arantes, Jonathan Lejeune, Madeleine Piffaretti, Olivier Marin, Pierre Sens, Julien Sopena, Alysson N. Bessani, Vinicius V. Cogo, Miguel Correia, Pedro Costa, Marcelo Pasin, and Fabrício A. B. Silva. Étude d'une architecture mapreduce tolérant les fautes byzantines. In *Actes des 20éme Rencontres francophones du parallélisme (RENPAR'11)*, May 2011. NAT LIP6 REGAL.

[3] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, March 2004.

[4] Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. Nephele/PACTs: a programming model and execution framework for web-scale analytical processing. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, pages 119–130, 2010.

[5] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa. DepSky: Dependable and secure storage in a cloud-of-clouds. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, pages 31–46, April 2011.

[6] Alysson N. Bessani, Vinicios V. Cogo, Miguel Correia, Pedro Costa, Marcelo Pasin, Fabricio Silva, Luciana Arantes, Olivier Marin, and Pierre Sens. Making hadoop mapreduce byzantine fault-tolerant. 2010, Faculdade de Ciências, Universidade de Lisboa and LIP6, Université de Paris, Lisboa and Paris, Portugal and France, 2010. Fast Abstract presented at the 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2010).

[7] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. In *ACM Transactions on Computer Systems, v. 20 n. 4*, pages 398–461, 2002, ACM New York, USA, 2002.

[8] Miguel Castro and Barbara Liskov. Practical Byzantine fault-tolerance and proactive recovery. *ACM Transactions Computer Systems*, 20(4):398–461, November 2002.

[9] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riché. UpRight cluster services. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles – SOSP'09*, October 2009.

[10] Miguel Pupo Correia and Paulo Sousa. Segurança no software. FCA editores, Lisboa, Portugal, 2010.

[11] Pedro Costa, Marcelo Pasin, Alysson N. Bessani, and Miguel Correia. Byzantine fault-tolerant mapreduce: Faults are not just crashes. Article submited in the 3rd IEEE International Conference on Cloud Computing Technology and Science (IEEE Cloudcom 2011).

[12] Flaviu Cristian. Understanding fault-tolerant distributed systems. volume 34, pages 56–78, 1993.

[13] Jeffrey Dean. Large-scale distributed systems at google: Current systems and future directions. Keynote speech at the 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware (LADIS), October 2009.

[14] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating Systems Design & Implementation*, December 2004.

[15] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative MapReduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 810–818, 2010.

[16] Jaliya Ekanayake, Shrideep Pallickara, and Geoffrey Fox. MapReduce for data intensive scientific analyses. In *Proceedings of the 2008 Fourth IEEE International Conference on eScience*, pages 277–284, 2008.

[17] Zacharia Fadika and Madhusudhan Govindaraju. LEMO-MR: Low overhead and elastic MapReduce implementation optimized for memory and cpu-intensive applications. In *Proceedings of the 2nd IEEE International Conference on Cloud Computing Technology and Science*, pages 1–8, 2010.

[18] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 29–43, 2003.

[19] Rachid Guerraoui and Andre Schiper. Consensus: The big misunderstanding. In *Proceedings of the 6th IEEE Workshop on Future Trends of Distributed Computing*

Systems, FTDCS '97, pages 183–, Washington, DC, USA, 1997. IEEE Computer Society.

[20] Thilina Gunarathne, Tak-Lon Wu, Judy Qiu, and Geoffrey Fox. MapReduce in the clouds for science. In *Proceedings of the 2nd IEEE International Conference on Cloud Computing Technology and Science*, pages 565–572, 2010.

[21] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 59–72, 2007.

[22] Z. M. Kedem, K. V. Palem, A. Raghunathan, and P. G. Spirakis. Combining tentative and definite executions for very fast dependable parallel computing. In *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, pages 381–390, 1991.

[23] Sriram Krishnan, Chaitanya Baru, and Christopher Crosby. Evaluation of MapReduce for gridding LIDAR data. In *Proceedings of the 2nd IEEE International Conference on Cloud Computing Technology and Science*, pages 33–40, 2010.

[24] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. In *ACM Transactions on Programming Languages and Systems, vol. 4 n. 3*, pages 382–401, 1982, ACM New York, USA, 1982.

[25] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programing Languages and Systems*, 4(3):382–401, July 1982.

[26] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, October 1998.

[27] Fabrizio Marozzo, Domenico Talia, and Paolo Trunfio. Adapting MapReduce for dynamic environments using a peer-to-peer model. In *Proceedings of the 1st Workshop on Cloud Computing and its Applications*, October 2008.

[28] Jean-Philippe Martin, Lorenzo Alvisi, and Michael Dahlin. Small byzantine quorum systems. pages 374 – 383, 2002, University of Texas at Austin, 2002.

[29] Jean-Philippe Martin, Lorenzo Alvisi, and Mike Dahlin. Minimal Byzantine storage. In *Proc. of the 16th International Symposium on Distributed Computing - DISC 2002*, pages 311–325, October 2002.

[30] Kirk McKusick and Sean Quinlan. GFS: evolution on fast-forward. *Communications of the ACM*, 53:42–49, March 2010.

[31] Mircea Moca, Gheorghe Cosmin Silaghi, and Gilles Fedak. Distributed results checking for MapReduce in volunteer computing. In *Proceedings of the 5th Workshop on Desktop Grids and Volunteer Computing Systems*, May 2011.

[32] Edmund B. Nightingale, John R. Douceur, and Vince Orgovan. Cycles, cells and platters: an empirical analysisof hardware failures on a million consumer PCs. In *Proceedings of the EuroSys 2011 Conference*, pages 343–356, 2011.

[33] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 1099–1110, 2008.

[34] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of ACM*, 27(2):228–234, April 1980.

[35] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24, 2007.

[36] Luis F. G. Sarmenta. Sabotage-tolerance mechanisms for volunteer computing systems. *Future Generation Computer Systems*, 18:561–572, March 2002.

[37] Fred B. Schneider. Implementing fault-tolerant service using the state machine aproach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[38] Bianca Schroeder and Garth A Gibson. Understanding failures in petascale computers. *Journal of Physics: Conference Series*, 78(1), 2007.

[39] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM errors in the wild: a large-scale field study. In *Proceedings of the 11th International Joint Conference on Measurement and Modeling of Computer Systems*, pages 193–204, 2009.

[40] Jason Venner. *Pro Hadoop (Expert's Voice in Open Source)*. Apress, 6 2009.

[41] Paulo Veríssimo and Luis Rodrigues. *Distributed Systems for System Architects*. Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[42] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung. EBAWA: Efficient Byzantine agreement for wide-area networks. In *Proceedings of the 12th IEEE International High Assurance Systems Engineering Symposium*, November 2010.

[43] Tom White. *Hadoop: The Definitive Guide*. O'Reilly, first edition edition, june 2009.

[44] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving MapReduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, pages 29–42, 2008.