UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

ALEXANDRE MIYAZAKI

Avaliação do modelo MapReduce em diferentes arquiteturas: um comparativo entre Hadoop e Maresia

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Cláudio Fernando Resin Geyer

Co-orientador: Prof. MSc. Pedro de Botelho Marcos

Porto Alegre, Julho de 2014

“The dictionary is the only place where success comes before work.”
— VINCE LOMBARDI

AGRADECIMENTOS

Inicialmente, agradecer ao professor Claudio Geyer pela oportunidade de realizar este trabalho no GPPD (Grupo de Processamento Paralelo e Distribuído). Ao professor Pedro de Botelho Marcos da FURG (Fundação Universidade Federal do Rio Grande) pelo suporte ativo, revisões rápidas e críticas construtivas durante todo o trabalho.

Agradecer também à minha mãe pelo apoio durante não só a faculdade, mas por toda a minha jornada de estudos. A minha namorada, agradecer pela compreensão nos momentos difíceis em que eu fiquei imerso em trabalho deixando-a de lado.

E finalmente, agradecer a todos aqueles que de alguma forma me ajudaram a concluir esta importante etapa.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

DFS	Distributed File System
DHT	Distributed Hash Table
GFS	Google File System
HDFS	Hadoop Distributed File System
MRBS	MapReduce Dependability and Performance Benchmarking
NFS	Network File System
P2P	Peer-to-peer
SPOF	Single Point of Failure

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

No mundo computacional, tem-se um aumento constante na demanda de processamento. Há cada vez mais dados a serem processados, de forma que frequentemente eles exigem uma abordagem distribuída para que isto ocorra em um tempo aceitável.

O MapReduce é um modelo de programação paralela, que visa facilitar o desenvolvimento deste tipo de aplicação, gerenciando grande parte dos complicadores como comunicação, tolerância a falhas, etc.. No Hadoop, *framework open source* mais utilizado pela comunidade científica que implementa este modelo, há dois pontos únicos de falha que podem comprometer toda a computação de um *job*.

Tendo isso como inspiração, foi desenvolvido em um trabalho anterior a este uma nova arquitetura para o MapReduce, denominada Maresia, além de um protótipo que a utiliza. Esta arquitetura segue um modelo *peer to peer* para processar *jobs*.

O objetivo deste trabalho é fazer uma avaliação aprofundada do desempenho do protótipo, analisando o seu comportamento e comparando seus resultados com testes executados em uma arquitetura Mestre/Escravo. Para um melhor comparativo foram implementadas novas funcionalidades durante o trabalho que permitem que ele tenha um fluxo de execução mais semelhante ao Hadoop.

Os resultados mostram um protótipo que sofre com alguns pontos ineficientes, principalmente a transmissão de dados intermediários. Além disso, o comparativo com o Hadoop mostra que esta nova arquitetura é bastante promissora, no entanto, seu gargalo precisa ser otimizado.

Palavras-chave: P2P, MapReduce, Hadoop, Maresia, SPOF, programação paralela.

ABSTRACT

In the computational world, has been a steady increase in demand for processing. There is an increasing amount of data to be processed, that often demand a distributed approach to occurs in an acceptable time.

MapReduce is a parallel programming model, which aims to ease the development of this kind of application, managing great part of the complicating factors like communication, fault tolerance, etc.. In Hadoop, the most used open source framework by the scientific community which implements this model, there are two single points of failure that can compromise the process of a job.

Having this as inspiration, a previous work has designed a new MapReduce architecture, called Maresia, besides an prototype to use it. This architecture follows a peer to peer model to process jobs.

The objective of this work is execute a thorough performance evaluation of the prototype, analyzing it's behavior and comparing the results with tests performed in a Master/Slave architecture. For a better comparison, new features has been implemented that allows it to have a workflow more similar to Hadoop.

The results shows a prototype that suffers with some inefficient points, mainly the intermediate data transmission. Furthermore, the Hadoop comparison with the model shows that this architecture is quite promising, however, the bottleneck must to be optimized.

Keywords: P2P, MapReduce, Hadoop, Maresia, SPOF, programação paralela.

1 INTRODUÇÃO

A evolução da tecnologia e a popularização dos sistemas computacionais (não apenas PCs, mas também dispositivos móveis) em associação com o contínuo crescimento de serviços disponibilizados para os usuários de tais sistemas, geram uma grande quantidade de dados. Como exemplo de fornecedor de serviços que se encaixa neste perfil temos o Facebook que em 2012 agregava por mês aproximadamente 7 petabytes de imagens (CORPORATE, 2012).

De acordo com uma pesquisa realizada pela IDC (International Data Corporation), de 2005 a 2020 o mundo digital deve crescer por um fator de 300, de 130 exabytes para 40 000 exabytes (40 trilhões de gigabytes) (GANTZ; REINSEL, 2012). Além disso, uma estimativa da IDC patrocinada pela EMC (EMC, 2013a) mostra que de Janeiro a Maio de 2013 foram gerados mais de 1200 exabytes (EMC, 2013b).

Esse grande volume de dados frequentemente precisa ser processado para ter algum valor. Tendo em vista a sua magnitude, a metodologia para processá-los não pode ser aquela convencional de sistemas sequenciais, pois o tempo para realizar a computação necessária seria proibitivo. Uma alternativa para isso seria realizar o processamento distribuído, no entanto, desenvolver aplicações para este tipo de ambiente não é uma tarefa trivial pois envolve considerar uma série de fatores relacionados à plataforma como tolerância a falhas, comunicação, sincronização, balanceamento de carga, entre outros.

Com o intuito de facilitar o processamento de grandes volumes de dados, o Google desenvolveu o MapReduce, um modelo de programação voltado à computação intensiva em dados para ambientes largamente distribuídos (DEAN; GHEMAWAT, 2008). Após o Google houveram diversas implementações do modelo em várias plataformas como *multicore* (Phoenix (RANGER et al., 2007)), *General-purpose computing on graphic processing units* (GPGPUs) (Mars (FANG et al., 2011)), *Cloud Computing* ((LIU; ORBAN, 2011) e (GUNARATHNE et al., 2010)), além da implementação para ambientes largamente distribuídos *open source* mais popular, o Hadoop (WHITE, 2009), mantido pela Apache Software Foundation (HADOOP, 2013).

Em relação às implementações para *clusters* de máquinas de prateleira (ambiente alvo do modelo original), o Hadoop é fortemente baseado em seu predecessor desenvolvido pelo Google. Nessas implementações, há uma grande preocupação com tolerância a falhas a qual o Google considera como um fator sempre presente e não como uma exceção. Porém ainda existem pontos únicos de falha (do Inglês *Single Points of Failure*) que carecem de uma solução eficiente em termos de alocação de recursos.

1.1 Motivação

Um ponto único de falha é definido como uma parte ou um componente que ao falhar compromete o funcionamento de todo o sistema. Em modelagens onde disponibilidade e confiabilidade são requisitos chave, esta característica não deve existir.

O MapReduce segue uma arquitetura Mestre/Escravo (do Inglês, *Master/Worker* ou *Master/Slave*). A tolerância a falhas garantida pela maioria das implementações do modelo (esta maioria inclui o Hadoop) é exclusivamente voltada para falhas nos *workers*. Além disso, o sistema de arquivos utilizado pelo modelo também segue esta arquitetura, que é resiliente a falhas apenas nos escravos. Portanto, identifica-se dois SPOFs (abreviatura de *Single Points of Failure*), o nó mestre do *job* e o *NameNode*[1] do DFS, anteriormente citados em (DEAN; GHEMAWAT, 2004).

Tendo em vista que, em caso de falha de algum dos SPOFs, todo o processamento realizado até o momento é perdido, isto pode ser um problema sério dependendo do contexto. Um bom exemplo é um usuário que aluga ciclos de máquina para executar seus *jobs* MapReduce. Caso haja uma falha de *crash* em algum SPOF, ele perderá toda a computação realizada até o momento, fazendo com que haja um prejuízo financeiro. Outro bom exemplo são alocações com limite de duração, onde o usuário estima que seu *job* irá finalizar em um determinado tempo, no entanto ocorre uma falha deste tipo, expirando sua alocação e fazendo com que ele não obtenha o resultado da computação desejada.

Diferentes abordagens foram propostas para a resolução do problema dos pontos únicos de falha no MapReduce. A primeira delas é baseada em replicação, onde nós redundantes são inseridos no sistema (WANG et al., 2009),(MAROZZO; TALIA; TRUNFIO, 2011). Nesta abordagem há um desperdício de capacidade computacional pois as redundâncias só serão efetivamente utilizadas em caso de falha do nó principal. A segunda opção de tratamento de SPOFs, trata exclusivamente falhas no *NameNode* do DFS (CLEMENT et al., 2009), (DATASTAX, 2013). Além destas duas abordagens, há o modelo de tolerância a falhas utilizado em Computação em Nuvem, o qual consegue resolver todos os SPOFs fazendo uso de serviços específicos destes ambientes (LIU; ORBAN, 2011).

Portanto verificou-se que o modelo necessitava de uma abordagem que auxiliasse o tratamento destes pontos únicos de falha. Esta abordagem deveria ser eficiente em termos de utilização de recursos, sem a necessidade de nós redundantes que seriam utilizados apenas em uma situação de falha do nó principal. Além disso, esta abordagem deveria ser aplicável em uma gama maior de ambientes, já que os mecanismos existentes que tratam todos os SPOFs são aplicáveis apenas em ambientes de Computação em Nuvem.

Tendo isso como motivação, recentemente uma nova abordagem em relação à arquitetura foi desenvolvida, denominada Maresia (MARCOS, 2013). Ela utiliza uma rede P2P (abreviatura de *Peer-to-Peer*) para descentralizar a informação contida nos SPOFs, dinstribuindo-a entre os *peers*.

Como a implementação da arquitetura Maresia foi avaliada apenas com o intuito de mostrar a viabilidade de execução e o tratamento de SPOFs do MapReduce em ambientes P2P, o trabalho corrente tem três principais objetivos:

- Realizar experimentações para avaliação do comportamento do protótipo, objetivando a identificação de gargalos.

[1] Denominação atribuída ao nó mestre do DFS

- Avaliar comparativamente o protótipo em relação ao Hadoop para verificar se a nova arquitetura proposta pode ser competitiva com o modelo mais utilizado (*Master/Worker*).

- Realizar uma análise simples da escalabilidade da arquitetura. Deve-se apenas executar o protótipo em um ambiente com um grande número de máquinas para avaliar seu comportamento.

Para realizar estas avaliações, deve-se implementar alguns mecanismos que foram simplificados no *framework*, os quais serão mostrados a seguir. A realização deste trabalho visa o valor agregado nesta nova arquitetura, a qual resolve os pontos únicos de falha de forma eficiente e pode ser aplicada nos ambientes onde o MapReduce é mais utilizado.

Nos resultados obtidos foi possível identificar os gargalos e os pontos que estão otimizados. Além disso, em alguns casos bem restritos, o protótipo foi mais eficiente que o Hadoop. Isso mostra que essa arquitetura tem o potencial de ser tão ou até mais eficiente que os modelos *Master/Worker*, oferecendo ainda o tratamento de pontos únicos de falha.

1.2 Organização

O restante deste trabalho segue a seguinte organização: No Capítulo 2 são apresentados detalhadamente o modelo MapReduce e o Hadoop. No Capítulo 3 descrevem-se a arquitetura Maresia e o *workflow* de *jobs* MapReduce sendo executados na mesma. No Capítulo 4 é detalhada a implementação do protótipo e citadas as modificações realizadas no mesmo. A metodologia dos experimentos e os resultados obtidos são apresentados no Capítulo 5 e no Capítulo 6 este trabalho é concluído e são elencados temas relacionados a trabalhos futuros.

2 MAPREDUCE

O MapReduce é um modelo de programação paralela desenvolvido pelo Google que objetiva o processamento de grandes volumes de dados em ambientes distribuídos. Ele foi proposto com o intuito de facilitar a programação de aplicações paralelas, sem que o desenvolvedor necessite preocupar-se com aspectos advindos da distribuição da computação como comunicação, tolerância a falhas e etc.. Este modelo baseia-se em duas primitivas presentes em linguagens funcionais: a primitiva de *Map* e a primitiva de *Reduce*. Utilizou-se desta abordagem pois notou-se que frequentemente era necessário mapear fragmentos de dados de entrada à uma chave identificadora e posteriormente realizar uma computação sobre os dados mapeados para uma mesma chave (DEAN; GHEMAWAT, 2008). Portanto, ao desenvolver uma aplicação MapReduce, o desenvolvedor precisa preocupar-se apenas em programar essas duas funções.

O restante deste Capítulo é destinado à descrição detalhada do modelo. Na Seção 2.1 são apresentados os passos básicos do funcionamento do sistema, na seção 2.2 é apresentada a arquitetura original (aquela proposta pelo Google), na Seção 2.3 são descritas otimizações e etapas internas realizadas durante a execução e finalmente a Seção 2.4 descreve detalhadamente o Hadoop comparando-o com o modelo original.

2.1 Visão Geral do Modelo

O *workflow* do MapReduce pode ser dividido em três fases bem definidas: A fase de mapeamento, a fase de redução e a fase de *Shuffle*. No mapeamento os dados de entrada são processados, gerando tuplas no formato <*chave,valor*>. Após esta fase ocorre o *Shuffle*, que será explicado em detalhes na Seção 2.3 e por último, ocorre a redução, que utiliza as tuplas geradas na fase de *Map* e realiza um processamento sobre conjuntos de valores que foram mapeados para a mesma chave. A Figura 2.1 mostra o fluxo de execução do MapReduce.

Primeiramente, temos um nó mestre (que executa esta função não só do modelo, mas também do *sistema de arquivos distribuído*[1] utilizado pelo *framework*) com dados locais, os quais irão ser utilizados de entrada para o *job* MapReduce. Quando estes são copiados para o DFS, há uma divisão da entrada (*Input splitting* [2] (VENNER, 2009)) em *chunks*[3], onde o tamanho destes blocos pode ser definido pelo usuário (geralmente 64 megabytes) nas configurações do sistema de arquivos. Após o *split* dos dados, os blocos são enviados para serem armazenados nos escravos do sistema de arquivos, enquanto o mestre

[1] Também conhecido como DFS, abreviatura de *distributed file system*

[2] *Split* dos dados de entrada ou Input Splitting é a partição dos dados de entrada em *chunks*.

[3] Um *chunk* é uma partição do arquivo de entrada.

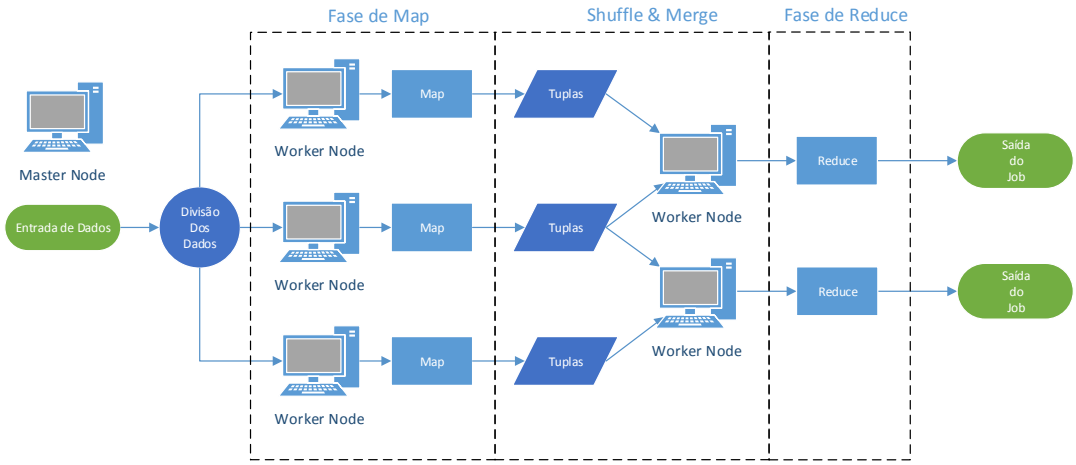

Figura 2.1: Funcionamento do modelo MapReduce.

armazena informações sobre a localização dos dados.

Com os *chunks* armazenados no DFS a fase de *Map* pode ser executada, resultando em conjuntos de pares intermediários. Eles sofrem um pré-processamento advindo da fase de *Shuffle* (detalhado na Seção 2.3) e finalmente são disponibilizados para que os *reducers* possam obtê-los. Após todos os dados intermediários serem distribuídos, os *workers* responsáveis pela tarefa de *Reduce* realizam um pré-processamento sobre as saídas da tarefa de *Map* (também advindo da etapa de *Shuffle*) e o *Reduce* é executado, gerando a saída do *job*.

É importante salientar que o número de tarefas de *Reduce* pode ser diferente do número de tarefas de *Map* (o número de reduções e configurável pelo usuário) e os nós que executam as tarefas de redução, não são necessariamente diferentes daqueles que executaram as tarefas de mapeamento. Portanto, um nó que executou uma tarefa de mapeamento pode (e geralmente é) responsável por uma tarefa de redução. Ao finalizar o *job*, os dados de saída gerados são armazenados no DFS.

2.2 Arquitetura

Em sua implementação original, o modelo segue uma arquitetura Mestre/Escravo. O nó mestre é encarregado de escalonar tarefas e detectar falhas (de tarefas ou de escravos). Já o conjunto de nós escravos tem a função de executar as tarefas de mapeamento e redução. Além do mestre e dos escravos, temos o módulo de armazenamento de dados, o qual consiste em um sistema de arquivos distribuído.

O DFS também utiliza essa arquitetura, sendo o mestre responsável por manter informações relativas aos dados armazenados e garantir sua disponibilidade, utilizando mecanismos como a replicação. Os nós escravos por sua vez, são responsáveis pelo armazenamento efetivo dos dados no sistema de arquivos. É importante salientar que os mesmos nós utilizados como nós escravos do DFS são nós escravos do MapReduce, permitindo ao modelo a exploração da localidade dos dados, evitando que estes necessitem ser transferidos.

Além de sua arquitetura original, foram desenvolvidas diversas implementações do modelo para ambientes variados. A Tabela 2.1 relaciona as implementações do modelo e

os ambientes para os quais elas foram desenvolvidas.

Implementação	Ambiente Alvo
Hadoop (WHITE, 2009)	Clusters
Phoenix (RANGER et al., 2007)	Multicore
Mars (FANG et al., 2011)	GPGPUs
Cloud Mapreduce (LIU; ORBAN, 2011)	Cloud computing
Azure Mapreduce (GUNARATHNE et al., 2010)	Cloud computing

Tabela 2.1: Relação de implementações do MapReduce com seu ambiente alvo.

2.3 Otimizações e Fase de *Shuffle*

A etapa intermediária denominada *Shuffle* é executada em conjunto com a transferência de dados dos *mappers* para os *reducers*. Ela pode ser separada em duas fases, uma realizada no nó de mapeamento e outra realizada no nó de redução.

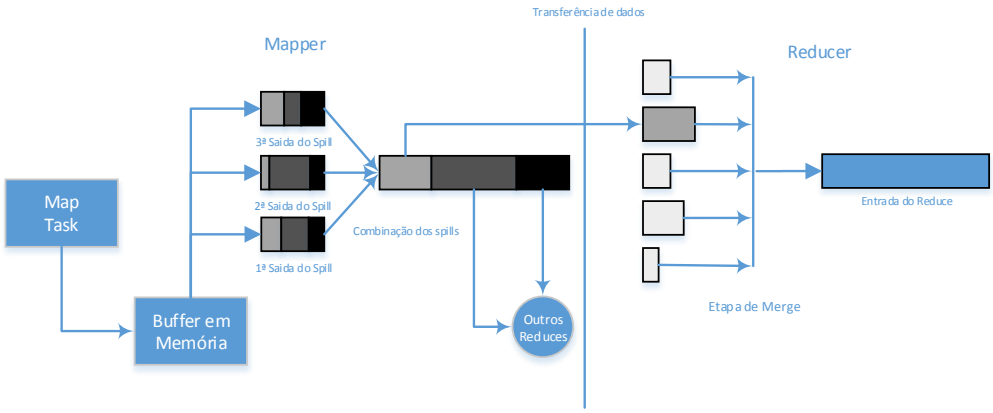

Figura 2.2: Fase de *Shuffle*.

A Figura 2.2 exemplifica com detalhes o fluxo de execução. A saída do *Map* não é simplesmente escrita em disco, cada uma destas tarefas tem um *buffer* circular, onde são escritas as suas saídas (o tamanho deste é configurável pelo usuário e o valor padrão é 100 megabytes). Quando o conteúdo de um destes chega a um limite (também definido pelo usuário e com valor padrão 80%), uma *thread* é executada em *background* e começa a realizar o *spill*[4] dos dados. As saídas da tarefa de mapeamento que está sendo executada são continuamente escritas no *buffer* circular e se este chegar a sua capacidade máxima a tarefa é bloqueada até que o *spill* seja completado.

Antes de escrevê-los em disco, a *thread* que realizará o *spill* faz uma divisão dos dados em partições correspondentes aos *reducers* que estes dados serão enviados. Na

[4]Neste contexto, *spill* dos dados refere-se à gravação dos dados em disco.

Figura, podemos verificar três divisões na saída dos *spills*. Dentro de cada divisão, ela realiza um ordenamento em memória com a chave da tupla como parâmetro.

Cada vez que o *buffer* circular excede o limite para executar o armazenamento das saídas da tarefa de *Map* para disco, um novo arquivo de *spill* é criado (Na Figura 2.2, pode-se visualizar três arquivos). Portanto, quando a tarefa de mapeamento acaba de escrever suas saídas, frequentemente temos vários arquivos os quais são combinados mantendo as propriedades dos arquivos originais (ordenados e particionados) para que a tarefa seja concluída. Após a combinação dos arquivos, o nó notifica o *Master* da localização dos dados, finalizando assim a fase de *Shuffle* no *mapper*.

Já no *reducer*, o mesmo começa a copiar os dados necessários para a execução de sua tarefa assim que os *Maps* vão sendo finalizados. No exemplo, apenas uma das divisões da saída da tarefa de mapeamento deve ser processada pela tarefa de redução em questão (as outras são enviadas para outros *reducers*). A medida que as entradas da tarefa de *Reduce* vão sendo copiadas para o *worker* responsável, uma *thread* vai realizando uma combinação destes dados visando adiantar trabalho a ser realizado na etapa de *Merge*[5]. Quando todas as saídas do *Map* necessárias para o *Reduce* estão disponíveis no *worker* (no exemplo temos cinco destas saídas), o *reducer* entra na fase de *Merge*, onde realiza a combinação das saídas da fase de *Map* mantendo a sua ordenação. Após realizada esta combinação, a entrada do *Reduce* está disponível para iniciar a tarefa.

Ainda tratando-se da fase de *Shuffle*, existe uma operação de otimização opcional que pode ser executada nesta fase, denominada *Combine*. Ela consiste em uma função definida pelo usuário que é executada na saída do *Map* e o objetivo desta é minimizar a transferência de dados entre os *mappers* e *reducers*. De acordo com a proposta inicial do MapReduce, é assumido que recursos de rede são escassos (DEAN; GHEMAWAT, 2008), o que torna esta otimização importante.

Uma otimização que não é atribuída ao *Shuffle* são as *Backup Tasks*. Foi observado que um dos fatores que mais causava lentidão na execução de um *job* eram nós que levavam um tempo proibitivo para completar algumas das últimas tarefas de *Map* (estes *workers* são denominados *Stragglers* (DEAN; GHEMAWAT, 2008)). Uma máquina pode tornar-se um *Straggler* por uma infinidade de motivos como escalonamento de outras tarefas na máquina que está executando o *job*, setores corrompidos em disco, etc..

Essas tarefas de *backup* funcionam da seguinte forma: Quando uma fase de mapeamento ou redução está chegando ao fim, o *framework* dispara cópias das tarefas que ainda não foram concluídas. Quando uma das cópias ou a original é terminada, esta tarefa é marcada como completada. De acordo com (DEAN; GHEMAWAT, 2008), este mecanismo melhora significantemente o desempenho do MapReduce, chegando a até 44% de redução no tempo de execução.

2.4 Hadoop

O Hadoop é um *framework* de código aberto que implementa o modelo MapReduce. Ele foi desenvolvido e atualmente é mantido pela Apache Software Foundation (HADOOP, 2013) e é um dos *frameworks* mais conhecidos que implementam o modelo. Esta implementação tem uma série de semelhanças em relação ao seu predecessor desenvolvido pelo Google.

Na literatura há alguns pontos onde as terminologias utilizadas nestas duas aplicações do modelo MapReduce diferem. Todavia, pode-se relacionar termos equivalentes, como

[5] O *Merge* é uma operação que faz parte do *Shuffle*.

mostra a Tabela 2.2.

Google MapReduce	Hadoop
Master	JobTracker
Worker	TaskTracker
Backup task	Speculative task
GFS Master	HDFS NameNode
GFS Chunkserver	HDFS DataNode

Tabela 2.2: Diferenças terminológicas entre Google MapReduce e Hadoop.

Os *TaskTrackers* enviam continuamente mensagens para o *JobTracker*. Estas mensagens, denominadas de *heartbeat*, contém informações sobre o *worker* além de sinalizar ao *JobTracker* que ele está ativo. Faz parte do conteúdo desta mensagem uma assinalação se o *worker* está apto para executar uma tarefa e se estiver, é tarefa do *Jobtracker* definir uma tarefa para ele.

Sobre o escalonamento de tarefas, os *Reduces* são escalonados seguindo uma política FIFO, visto que não há localidade de dados (as saídas do *Map* que por sua vez são as entradas do *Reduce* estão distribuídas entre os nós do ambiente). Todavia as tarefas de mapeamento são escalonadas explorando a localidade de dados. A decisão de escalonamento é tomada sempre designando tarefas onde a entrada esteja mais próxima do nó, sendo o ideal que os nós processem apenas tarefas as quais eles possuem os *chunks* de dados de entrada localmente.

O funcionamento segue os moldes do descrito anteriormente na Seção 2.1, relativo à implementação do Google.

2.4.1 Hadoop Distributed File System

Como o Hadoop costuma lidar com dados geralmente muito grandes (na ordem de gigabytes ou terabytes) e por ser uma abordagem de computação distribuída, torna-se necessário gerenciar os dados de entrada de uma forma diferente da convencional. Sendo um sistema de arquivos distribuído, um tipo mecanismo que gerencia o armazenamento de dados entre uma rede de máquinas, ele torna-se ideal para o modelo MapReduce.

O HDFS (abreviatura de Hadoop Distributed File System), é o sistema de arquivos distribuído utilizado pela implementação da Apache do modelo MapReduce. Ele é baseado no GFS (abreviatura de Google File System), utilizado na implementação do Google. Para exemplificar a capacidade do HDFS, de acordo com (SHVACHKO et al., 2010), os *clusters* que são utilizados no Yahoo! armazenam 25 petabytes nele. Além do grande volume, também foi levado em consideração no desenvolvimento deste componente que os dados armazenados depois de escritos não são mais modificados, sendo realizadas apenas leituras sobre os mesmos.

Além disso, o HDFS é desenvolvido para ser executado, assim como o Hadoop e o Google MapReduce, em *clusters* com centenas a milhares de máquinas de prateleira, onde a probabilidade de falhas é grande (DEAN; GHEMAWAT, 2008). Sendo assim, ele conta com mecanismos de tolerância a falhas para que haja uma alta disponibilidade dos dados armazenados.

Assim como em sistemas de arquivos convencionais, há o conceito de blocos que consiste na quantidade mínima de dados que pode-se ler ou escrever. Enquanto nestes eles têm tipicamente poucos bytes, o tamanho dos blocos no HDFS é bem maior sendo

definido como padrão 64 megabytes (configurável pelo usuário por arquivo armazenado), com o objetivo de minimizar o custo de *seeks*. Esta abstração traz ao DFS uma série de vantagens, dentre elas:

- Como os arquivos são divididos em blocos e estes são armazenados em diferentes máquinas, os arquivos armazenados podem ser maiores que qualquer disco no ambiente.
- Simplifica o gerenciamento do armazenamento de dados.
- Facilita a replicação, necessária para garantir a disponibilidade dos dados em um sistema de arquivos distribuído.

A arquitetura do HDFS compreende dois tipos de nós, organizados em um padrão Mestre/Escravo, onde *NameNode* realiza o papel de nó mestre do mecanismo e os *DataNodes* realizam o papel de escravos. É de responsabilidade do *NameNode* gerenciar o sistema de arquivos, mantendo os metadados relacionados a todos os arquivos e diretórios, saber onde todos os blocos de um determinado arquivo estão armazenados, gerenciar as réplicas que garantem a disponibilidade de dados (parâmetro configurável, com valor padrão de três réplicas), etc.. Por sua vez, os *DataNodes* armazenam efetivamente os blocos dos arquivos no HDFS.

Como citado anteriormente, o *NameNode* é um ponto único de falha e o próprio Hadoop oferece um mecanismo para torna-lo tolerante a falhas. Este mecanismo consiste em executar um *NameNode* secundário, que mantem uma cópia da imagem do *Namespace* (hierarquia de arquivos e diretórios) que pode ser usada em caso de falha no principal. Todavia esta abordagem quase sempre resulta em uma perda de dados pois o estado do *NameNode* secundário esta geralmente atrasado em relação ao primário.

2.4.2 Tolerância a Falhas

Como na implementação original, o Hadoop trata falhas como se fossem regra e não como uma exceção. Ele implementa mecanismos muito semelhantes ao *framework* do Google, onde a reexecução é a principal técnica utilizada.

Em relação à falhas de tarefas, o *TaskTracker* registra que a mesma falhou armazenando alguma informação nos *logs* de usuário e libera o *slot*[6] de tarefa (mapeamento ou redução) para a execução de uma nova. A notificação desta falha ao *JobTracker* é realizada através de *heartbeat*.

Quando o *JobTracker* recebe a notificação de que a tarefa escalonada para o *worker* falhou, ele irá reescalona-la tentando evitar atribuí-la ao mesmo nó. Além disso, se uma tarefa falha mais do que quatro vezes (valor padrão que pode ser configurado), ela não é mais escalonada, e o *job* é abortado indicando erro de execução.

Em alguns tipos de aplicações, pode ser tolerado que algumas tarefas falhem e independente disto, ter um resultado satisfatório. Para este tipo de situação, pode ser configurado uma porcentagem de falhas que podem ocorrer sem que o *job* seja abortado, onde *Maps* e *Reduces* são configurados separadamente.

Quando ocorre uma falha de *crash* ou quando a máquina torna-se muito lenta (vários fatores podem contribuir para isto como o escalonamento de processos não relacionados

[6] *Slots* são *threads* de processamento. Eles podem ser de mapeamento (*thread* que executa tarefas de *Map*) ou redução (*thread* que executa tarefas de *Reduce*).

com o MapReduce, mal funcionamento de hardware, etc.), o *TaskTracker* envia mensagens de *heartbeat* ao *JobTracker* muito infrequentemente, se é que não para de enviá-las. Se o *JobTracker* não recebe uma mensagem de *heartbeat* dentro de um intervalo que por padrão é configurado para dez minutos ele retira esta máquina de sua lista de *workers* aptos a receberem uma tarefa.

O *JobTracker* ainda reescalona as tarefas que foram atribuídas ao *TaskTracker* que não é mais apto a receber tarefas, inclusive aquelas que já haviam sido completadas. Isto se deve pois os pares intermediários resultantes da fase de *Map* são armazenados nos discos locais dos *workers* e portanto, quando ocorre uma falha de *crash* em um deles, estes dados podem tornar-se indisponíveis.

Um *TaskTracker* pode ainda ser colocado em uma *blacklist*, que é gerenciada pelo *JobTracker*. Eles são inseridos nesta lista quando o número de tarefas que falharam nele é consideravelmente maior que a média de falhas de tarefas no *cluster*. Para retirá-lo desta lista, ele deve ser reiniciado.

Como citado anteriormente, o *JobTracker* é outro ponto único de falha. O Hadoop, assim como o Google MapReduce, não implementa mecanismos de tolerância a falhas no *JobTracker*.

3 MARESIA

Maresia é a nomenclatura adotada para a arquitetura desenvolvida em (MARCOS, 2013). Ela tem como objetivo executar *jobs* MapReduce realizando não só o tratamento de falhas nos *workers*, mas também prover um sistema livre de SPOFs. Para isso é utilizada uma abordagem inspirada no Chord (STOICA et al., 2001), fazendo uso de sua arquitetura arquitetura P2P para descentralizar a informação contida nos pontos únicos de falha.

O restante deste capítulo é organizado da seguinte forma: Na Seção 3.1 é apresentada uma visão geral do Chord. Na Seção 3.2 é apresentada a arquitetura Maresia e finalmente, na Seção 3.3, é exemplificado o fluxo de execução de *jobs* MapReduce utilizando esta arquitetura.

3.1 Chord

O Chord é um protocolo que define como localizar chaves, inserir novos *peers* e tratar falhas em um ambiente P2P. Em relação à organização dos nós, eles são dispostos em forma de anel unidirecional. No processo de inicialização, Chord assume que os *peers* conseguem obter o IP de algum nó que esteja no anel. Depois de conectar-se com o *peer*, ele requisita o seu identificador, e após recebê-lo conecta-se à sua posição no anel (definida pelo id recebido). Quando o novo nó finaliza sua inserção, há a redistribuição de chaves e dos dados mapeados para estas chaves. Note-se que quando um *peer* deixa o anel, as conexões são reorganizadas e tanto as chaves quanto os dados são redistribuídos.

Tratando-se da atribuição de identificadores aos nós, este processo é realizado com uma função de *hash*, utilizando como parâmetro o endereço IP. Cada *peer* é responsável por um conjunto de chaves, no intervalo que vai do seu identificador até o identificador do próximo *peer* menos um. Para melhor entendimento, a Figura 3.1 mostra um exemplo de anel do Chord. Como pode-se visualizar, o nó de identificador 9 é responsável pelas chaves no intervalo de 9-14, enquanto o *peer* 15 é responsável pelas chaves 15-17.

O gerenciamento de dados é realizado seguindo uma técnica de DHT (abreviatura de *Distributed Hash Table*). Ou seja, para o armazenamento de um dado aplica-se a função de *hash* sobre o mesmo, gerando uma chave. Com esta chave, é possível encontrar o responsável por armazenar esta informação. Para obter um dado, a mesma função é aplicada sobre alguma informação relativa a ele gerando, como no armazenamento, uma chave. Com esta é possível fazer a requisição (de armazenamento ou recuperação) do dado para o nó responsável por armazená-lo.

No processo de localização do responsável por uma chave (necessário tanto no armazenamento quanto na recuperação), o requisitante verifica se o seu vizinho é o responsável pela chave. Se esta estiver no seu conjunto de chaves, a requisição é aceita e os dados são enviados (ou recebidos, em caso de armazenamento), caso contrário a requi-

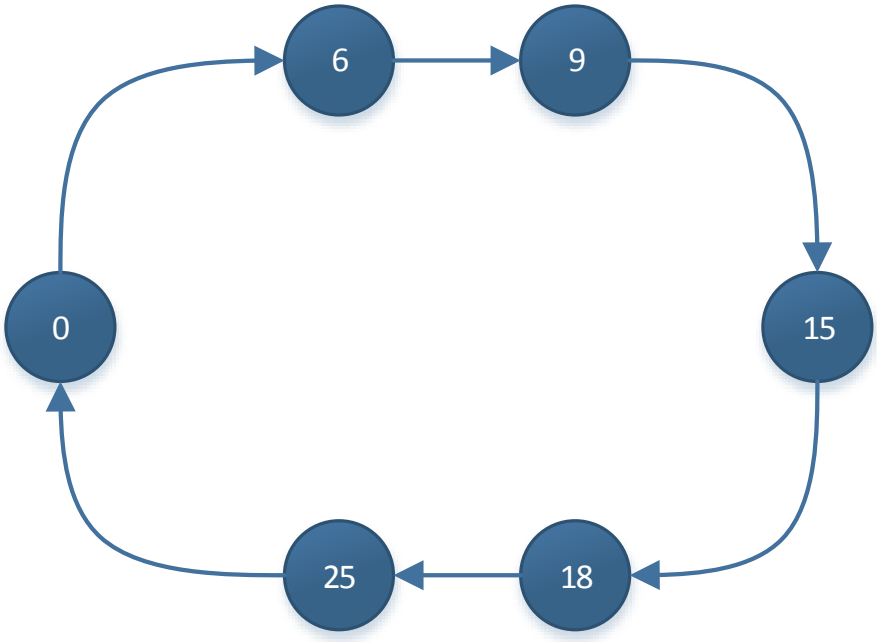

Figura 3.1: Organização dos nós em anel unidirecional utilizado pelo Chord.

sição é repassada ao próximo nó.

Note que esta técnica é ineficiente: pode ser preciso perguntar a todos os *peers* para encontrar aquele responsável por armazenar a informação desejada. Entretanto, o Chord otimiza esta busca utilizando *Finger Tables*. Estas tabelas contém informações sobre até *m peers*, onde *m* é o número de bits utilizados para a geração dos identificadores, e são atualizadas sempre que um nó entra ou sai do anel. Com a utilização da *Finger Table*, não há mais a restrição de que requisições devem ser realizadas apenas para vizinhos, ou seja, o *peer* pode realizar esta verificação diretamente com aquele que tem o identificador mais perto da chave desejada. Com esta otimização, o mecanismo de busca em questão cai de uma complexidade linear para complexidade logarítmica.

3.2 Maresia

Maresia é uma arquitetura P2P, baseada no Chord, que visa a execução de *jobs* no modelo MapReduce. Por tratar-se de uma arquitetura deste tipo, ela permite que sejam eliminados os pontos únicos de falha de forma eficiente (sem a adição de nós de backup), distribuindo as responsabilidades que eram delegadas aos SPOFs sobre os *peers* do ambiente. Sua principal motivação é a remoção dos SPOFs existentes no modelo Cliente/Servidor.

Para uma avaliação mais realista do trabalho, teve-se o cuidado de não acrescentar restrições de ambiente de execução. Sendo assim a arquitetura apresenta apenas as condições de execução herdadas dos modelos em que se baseia, são elas:

- Não são inseridos nodos durante a execução de um *job* (MapReduce).
- Não há comportamento Bizantino (MapReduce).
- Há um mecanismo externo para realizar o *bootstrap* dos *peers* (Chord).

Maresia é resiliente não só às falhas que são toleradas pelo Hadoop, mas também àquelas que podem ocorrer no *Master Node* e no *NameNode* do HDFS. Portanto, elimina-se dois pontos únicos de falha que podem ser gravemente prejudiciais ao *framework*.

Sendo inspirado no Chord, Maresia tem uma arquitetura bastante similar. Como pode-se verificar na Figura 3.2, ele também é composto de um conjunto de *peers* organizados em um anel. Também é utilizada uma DHT para atribuir uma informação a um peer. No entanto, diferente de seu inspirador, Maresia utiliza um anel bi-direcional. Esta característica foi adotada para auxiliar nos mecanismos de tolerância a falhas.

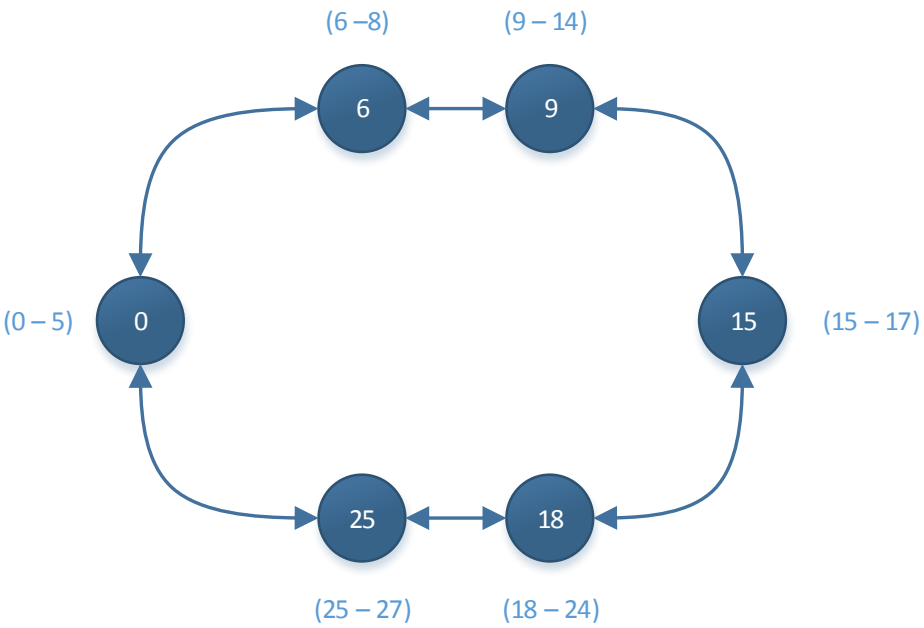

Figura 3.2: Organização dos *peers* em um anel bi-direcional no Maresia.

No anel, cada nó tem um identificador único e um intervalo numérico (que é composto do seu identificador até o identificador de seu vizinho menos um). Este intervalo define as chaves pelas quais o *peer* é responsável.

3.3 Execução de *jobs* MapReduce

Primeiramente, temos a inicialização dos mecanismos de controle da arquitetura. O anel é montado e a cada *peer* é atribuido o seu identificador seguindo o que fora citado na Seção 3.2. Como dito anteriormente, Maresia assume que há um mecanismo externo para realizar o processo de *bootstrap*. Caso esse mecanismo falhe, novos *peers* não podem

entrar no anel. Todavia, o *job* pode ser executado com aqueles que já pertencem ao mesmo.

Tendo sido distribuídos os identificadores e definidos os intervalos de chave que cada um é responsável, a distribuição de dados é realizada. Este processo é bastante similar aquele realizado no Hadoop, exemplificado na sessão 2.2. Ou seja, no arquivo de entrada (que pode ter vários gigabytes) é realizado um *split* em *chunks* de menor tamanho, que são enviados para os *peers*. Uma DHT é utilizada para definir o destinatário de cada um deles.

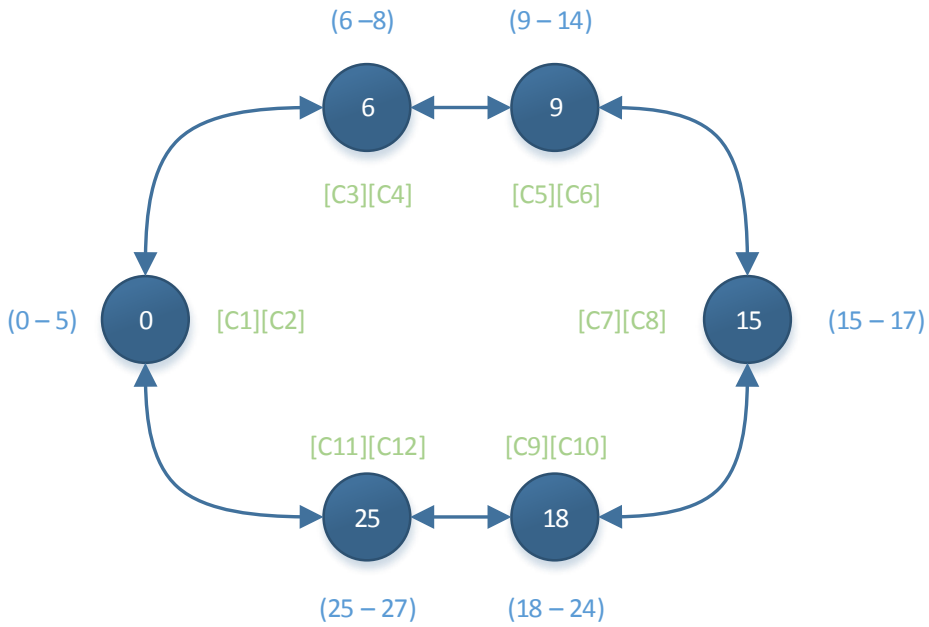

Figura 3.3: Anel após a distribuição dos dados.

Para exemplificar o estado do ambiente neste ponto, será utilizado o anel adotado nas Seções anteriores. Na Figura 3.3, tem-se o estado após o *split* e a distribuição dos dados. Verifica-se que da entrada originou 12 chunks, numerados de 1 a 12. A função de *hash* aplicada ao *chunk* 5 por exemplo, resultou em um valor entre 9 e 14, mapeando-o para o *peer* de identificador 9. Por sua vez, aplicada ao *chunk* 9, o resultado obtido foi entre 18 e 24, mapeando assim para o nó 18. Neste ponto, como Maresia não possui escalonamento dinâmico (visto que, não há um *Master Node*), cada um processa aquilo que lhe foi atribuído.

Após o processamento de um *chunk*, cada par intermediário deve ser enviado para o reponsável pelo seu *Reduce*. Para definir o destinatário, a função de *hash* é aplicada a cada chave do par e enviada para o responsável. Neste ponto, temos uma diferença de fluxo de dados em relação ao modelo original do MapReduce, onde os *reducers* que são responsáveis por obter os pares intermediários que irão processar. Este mecanismo simplifica e otimiza a execução, evitando assim que necessite-se de uma metodologia para encontrar os dados de entrada para a fase de redução.

Adicionalmente, para que o *Reduce* possa iniciar sua execução, todos os pares intermediários já devem se encontrar nos responsáveis pelo seu processamento (portanto, a fase de mapeamento deve estar completa). Como não há um nó central para alertar todos no ambiente de que a próxima fase pode iniciar, existe a necessidade de um mecanismo para a notificação de troca de fase. Este é exemplificado na Figura 3.4.

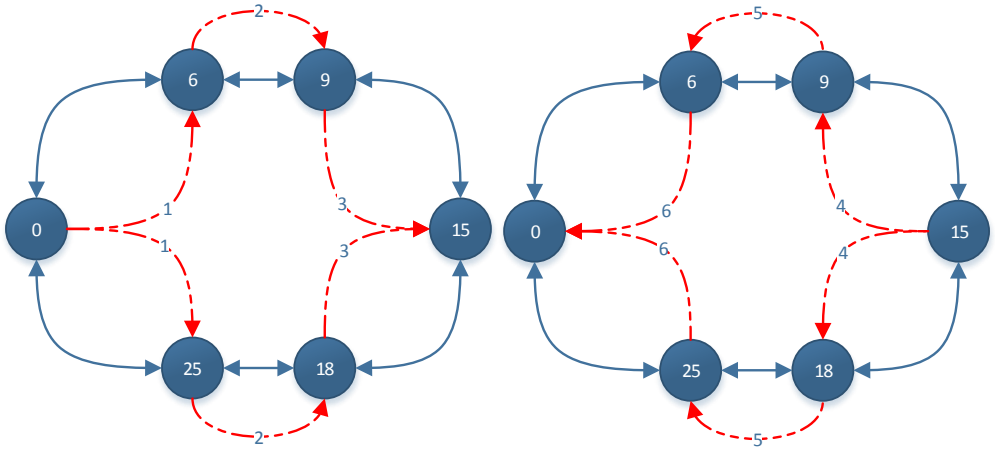

Figura 3.4: Passagem de *token* sinalizando o término do mapeamento.

Quando o nó originador (responsável pelo envio de tokens) finaliza a execução de todas as tarefas de mapeamento, ele envia dois *tokens* (um para cada vizinho), com o intuito de notificar o término da fase de mapeamento à todos no ambiente. Como exemplificado na Figura à esquerda, o *peer* com identificador 0 envia um *token* para cada vizinho [1]. Este deve ser repassado para o próximo vizinho deste no anel [2], no entanto ele só executa esta operação após o término de todas as tarefas de *Map* que lhe foram atribuídas. No momento em que um nó recebe dois *tokens* [3], ele pode iniciar a fase de redução. É importante salientar que eles não precisam estar presentes no *peer* no mesmo momento (como ocorre em [4] e [5]) para que ele seja notificado. Quando o nó originador receber ambos os *tokens* novamente, todos estarão executando a fase de *Reduce*.

Iniciada a fase de redução, cada um processa as tarefas relativas as chaves que lhe foram atribuídas. Por exemplo, se a um peer foram atribuídas as chaves 1,2 e 3, este irá executar três tarefas.

3.3.1 Tratamento de falhas

Até agora, foi vista a execução de *jobs* em um ambiente sem a presença de falhas. Lembrando que o objetivo desta nova arquitetura é não só ser resiliente a todas aquelas falhas cobertas pelo modelo Cliente/Servidor, mas também evitar alguns SPOFs presentes neste.

Começando pela replicação de dados, diferente do Hadoop e do Google MapReduce, Maresia não funciona sobre um DFS. Ou seja, a disponibilidade de dados deve ser garantida pelo próprio sistema. Portanto cada *peer* replica seus *chunks* em ambos os seus vizinhos no anel. Tal prática permite que o sistema seja resiliente a até duas falhas em um mesmo *chunk*, podendo ser extendida para a replicação ocorrer sempre que um *peer*

falhar.

Além disso, deve-se considerar aspectos relativos a falhas de tarefas e de *workers*. A detecção destas, assim como no Hadoop, é realizada por um mecanismo de *heartbeat*. Um *peer* periodicamente envia tais mensagens para ambos os seus vizinhos. Nota-se aqui, que elas são detectadas localmente, diferentemente dos modelos Cliente/Servidor do MapReduce, nos quais o *Master Node* é quem detecta todas as falhas do ambiente (pois é ele quem recebe as mensagens dos *peers*).

Os *heartbeats* além de sinalizarem que um *peer* está ativo, carregam uma série de informações, tais como a tarefa corrente, informações de progressão, informação sobre *token*, etc.. Eles também são utilizados para sinalizar os vizinhos de término de tarefas tanto de mapeamento quanto de redução. Isso permite ao ambiente, saber quais tarefas ainda não foram executadas, caso um nó sofra um *crash*.

Em relação a falhas nos *tokens*, a primeira que deve ser considerada é o *crash* no *peer* responsável por criá-los. Ela é tratada pela simples redefinição do *peer* que irá criar os *tokens*. Sendo assim, aquele que for o responsável pela chave 0 após a reestruturação do anel, será o novo responsável pela criação. Para detectar falhas em *token*, podem ser utilizados algoritmos apresentados em (ARANTES; SOPENA, 2010).

Devido à modificação do fluxo de dados no final do mapeamento (que são enviados dos *mappers* para os *reducers*, diferentemente das implementações mais tradicionais do MapReduce, onde o *reducer* requisita esses dados), e do fato de não haver um *Master Node* no ambiente, falhas em um *reducer* são complexas de tratar-se. Isso se deve ao fato de que esses pares intermediários que foram enviados para esta tarefa de *Reduce* são perdidos quando o nó falha. Para contornar isso, Maresia propõe um mecanismo de persistência de dados intermediários. Há diversas formas de implementação deles e todos têm como base a replicação dos dados de entrada do *Reduce* (similar à replicação de *chunks*), em ambos os vizinhos. Os dados podem ser replicados integralmente (alto custo, eficiente na recuperação de falhas) ou parcialmente (baixo custo, não tão eficiente na recuperação de falhas).

4 IMPLEMENTAÇÃO DA ARQUITETURA MARESIA

Maresia foi apresentado no Capítulo anterior, como uma nova abordagem para a execução de *jobs* MapReduce. Contudo, o modelo descrito precisa também ser validado. Para isso o trabalho original desenvolveu um protótipo que modela a arquitetura descrita. Serão apresentadas tanto a sua implementação original quanto as modificações realizadas durante este trabalho com o intuito de agregar mais valor à avaliação.

É importante destacar que as modificações desenvolvidas objetivam realizar uma melhor avaliação, tendo uma variabilidade maior de fatores, adicionando funcionalidades presentes no Hadoop e fazendo com que seus fluxos de execução sejam mais semelhantes. Estas aprimorações não foram incluídas no trabalho que propôs o modelo pois o foco daquele era mostrar a viabilidade de executar-se o MapReduce nesta nova arquitetura e de mostrar que os SPOFs eram de fato evitados.

Este Capítulo é organizado da seguinte forma: Na Seção 4.1, é descrita a implementação original, realizada em (MARCOS, 2013), e na Seção 4.2 são mostradas as modificações realizadas e as otimizações implementadas.

4.1 Implementação Original

O protótipo da arquitetura Maresia foi implementado em linguagem C. Para a comunicação entre *peers*, são utilizados *sockets* TCP e para o controle de *threads* são utilizadas *pthreads*. Não há nenhum tipo de API para desenvolver aplicações (funções de *Map* e *Reduce*), elas são codificadas diretamente no fonte do protótipo.

Quanto a fases da execução, ele segue os mesmos moldes das implementações Cliente/Servidor, adicionando-se a transmissão de *token*. Portanto, se tem a distribuição de dados, a fase de mapeamento (que inclui o *Combine*), a transmissão de dados intermediários, a passagem de *token* e finalmente a fase de redução. Três principais *threads* compõem a execução, são elas:

- *Thread* de mensagens: Responsável pela recepção de dados intermediários. Tanto os pares que são destinados ao *Reduce* executado neste *peer* quanto aqueles que são apenas réplicas (que serão processadas em seus vizinhos) são recebidos por esta *thread*. Ela aguarda por uma requisição de conexão TCP e quando a recebe, cria uma nova *thread* unicamente para receber os dados referentes a esta requisição.

- *Thread* de *Heartbeat*: Envia e recebe mensagens de *heartbeat*. Tais mensagens são utilizadas apenas como *alive* pois não há nenhum conteúdo relativo ao progresso de tarefas, tarefas já concluídas ou algo relacionado. Isso foi uma simplificação da implementação do protótipo.

- *Thread* de processamento: Responsável pelo processamento do *job* MapReduce em si. Ela deve iniciar as fases de mapeamento e redução.

A distribuição de dados é considerada uma operação disjunta do *job* MapReduce em si. O usuário deve explicitamente sinalizar ao protótipo por meio de um parâmetro que quer distribuir dados. Esta abordagem foi adotada pois os *frameworks* MapReduce não consideram a distribuição como parte do *job*. No Hadoop, por exemplo, deve-se colocar as entradas no HDFS via um comando específico para isto e então inicializar o *job* MapReduce.

Essa operação foi simplificada em comparação com aquela descrita na arquitetura Maresia. Ao invés de utilizar uma *hash* para mapear cada *chunk* para um nó, utiliza-se um algoritmo *round robin*, visto que a avaliação será dada por testes sobre um *cluster* de máquinas homogêneas, onde cada nó deverá receber a mesma quantidade de dados para serem processados. Note que esta simplificação não exclui a possibilidade da distribuição de dados ser realizada com uma função de *hash*, foi utilizada esta abordagem por simples facilidade de implementação. Com o anel montado e os dados distribuídos, pode-se inicializar a execução.

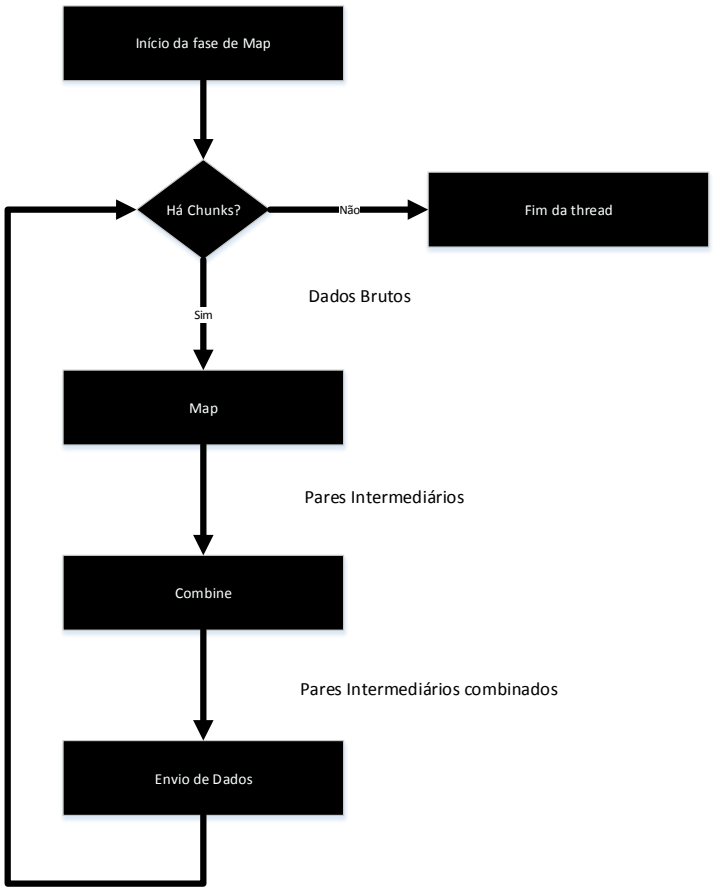

Figura 4.1: Fluxo de execução da fase de *Map*.

Não é possível executar duas ou mais tarefas de mapeamento em paralelo no *peer*, pois no processamento são utilizadas listas globais (para o armazenamento da saída da tarefa) sem um controle de acesso. Ou seja, a característica de *slots* não foi incluída nesta

primeira versão do protótipo. Como ilustrado na Figura 4.1, inicialmente há a execução da função de mapeamento, seguido do *Combine* e o envio dos dados.

É importante destacar que, diferente do Hadoop e do Google MapReduce, os dados intermediários resultantes da função de mapeamento não são persistidos em disco, sendo tratados sempre em memória até o seu envio. Isso resulta em um tratamento mais ágil dos dados intermediários (visto que estão todos em memória). No entanto, se as saídas dos mapeamentos em conjunto com dados de outros processos sendo executados no *peer* for grande o bastante para encher a memória, resultará em utilização de *swap* em disco, fazendo com que a tarefa demore mais para ser concluída.

Verifica-se que este fluxo pode ser levemente otimizado. Na execução do envio de dados, o processamento fica bloqueado aguardando essa transmissão. Esse processo pode ser, por exemplo, delegado para transmissão em uma outra *thread*, fazendo com que o *framework* possa seguir com a execução de tarefas.

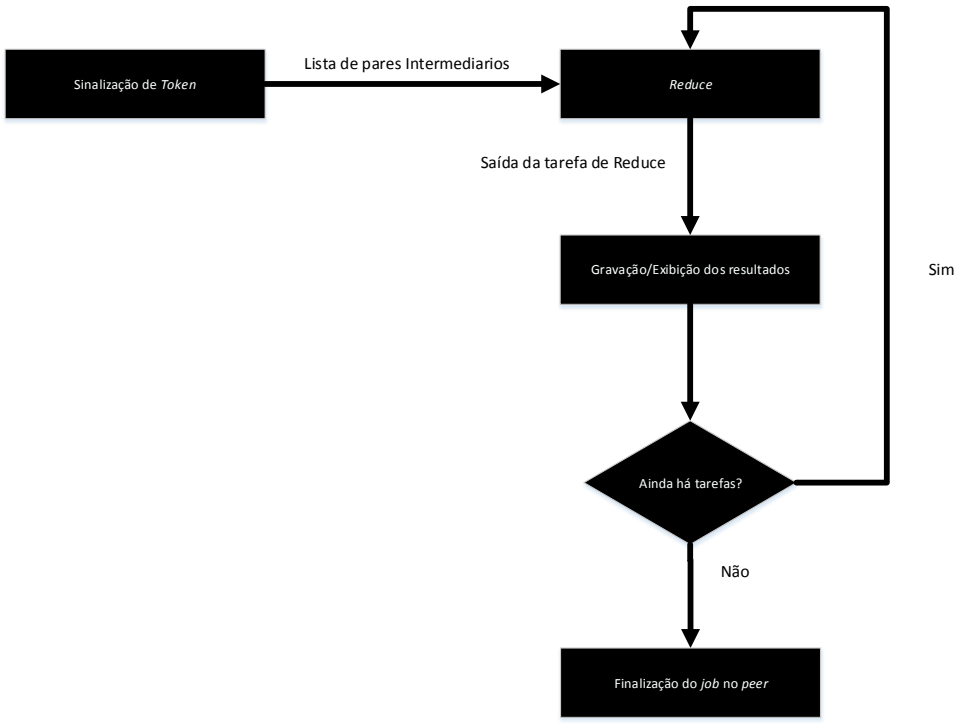

Figura 4.2: Fluxo de execução da fase de *Reduce*.

Após o término da fase de *Map*, o *peer* originador envia os *tokens* exatamente como descrito no Capítulo anterior, ou seja, um *token* para cada vizinho. Na Figura 4.2 pode-se verificar o *workflow* da fase de *Reduce*. Primeiramente temos a sinalização de que pode-se começar a processar as tarefas. Então inicializa-se uma tarefa de redução, gerando os resultados e gravando-os em arquivo ou exibindo-os para o usuário. Como a funcionalidade de executar-se tarefas em *slots* também não foi implementada para esta fase (também são utilizadas listas globais sem controles de acesso), se o *peer* é responsável por mais de uma tarefa, elas são executadas sequencialmente. Na finalização das tarefas de *Reduce*, são exibidos os resultados do processamento. Quando todos os *peers* acabam suas tarefas de redução, o *job* está terminado.

4.2 Modificações

Objetivando testes comparativos com um número maior de fatores a serem avaliados, foram realizadas algumas modificações no protótipo com o intuito de deixá-lo mais semelhante com o fluxo de execução seguido pelo Hadoop. Inicialmente, tem-se quatro *threads* ativas e não três como tinha-se na versão original do protótipo. Esta quarta *thread* tem o objetivo de transmitir pares intermediários resultantes das tarefas de *Map*.

Na Figura 4.3 é exibido um fluxograma da execução da fase de mapeamento. A primeira diferença a notar-se é a execução de tarefas em paralelo. No exemplo apresentado, tem-se dois *slots*, porém pode-se configurar este número para qualquer valor. O protótipo já criava os *slots*, todavia, como eram utilizadas listas globais para armazenamento de resultados do mapeamento, não era possível utilizar esta funcionalidade (pois tinha-se problemas de acesso concorrente a esta variável).

Para simplificar a implementação, eliminou-se a utilização de listas globais. Cada tarefa, trabalha agora com uma lista local (elas podem ter sua própria lista de resultados pois não há uma interdependência entre mapeamentos). Esta abordagem também contribuiu para o *Combine*, resultando em uma fase de *Map* onde as tarefas podem ser executadas paralelamente, de acordo com a configuração do usuário.

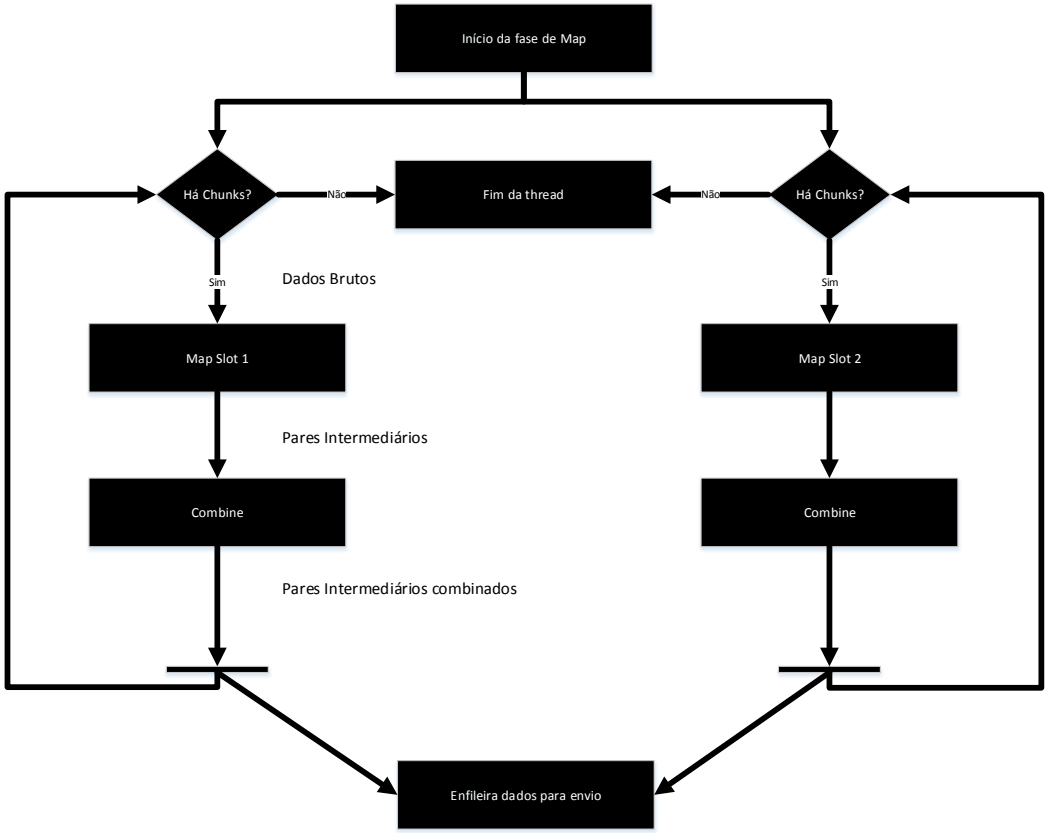

Figura 4.3: Modificações no fluxo de execução da fase de *Map* no protótipo.

Foi realizada uma otimização no envio dos dados resultantes da fase de *Map*. O fluxo de execução não é mais bloqueado pela transferência de dados. No término da execução de um mapeamento, seu resultado é enfileirado para envio, que é realizado pela *thread* que foi criada especificamente para isso, tornando todo o processo mais eficiente.

Em relação à fase de *Reduce*, havia os mesmos problemas apresentados no *Map*, ou seja, havia a criação de *slots* só que estes trabalhavam em cima de variáveis globais que apresentavam problemas de acesso concorrente. Para manter o padrão, o mesmo tipo de tratamento foi implementado, fazendo com que cada tarefa trabalhe com suas próprias variáveis. A execução é similar à fase de mapeamento, onde temos agora várias tarefas de redução sendo processadas em paralelo.

É importante salientar que não foi modificada a forma como o protótipo gerencia as *threads* (tanto de mapeamento, quanto de redução). Ou seja, apenas permitiu-se que tarefas (tanto de *Map* quanto de *Reduce*) sejam executadas em paralelo e que não haja um bloqueio do processamento pelo envio de dados. Portanto, as modificações realizadas são discretas, permitindo que sejam analisados os gargalos que já existiam na versão original do protótipo, mesmo que seu fluxo de execução tenha sido levemente alterado para melhor assemelhar-se com o Hadoop.

4.3 Considerações Finais

Este Capítulo definiu detalhadamente como o protótipo implementa a arquitetura Maresia. Foram sumarizadas as modificações implementadas que por sua vez paralelizam o fluxo de execução do protótipo, além de otimizar o processamento evitando seu bloqueio durante a transmissão de dados intermediários.

Um dos principais objetivos do trabalho é avaliar o desempenho *framework* que implementa a arquitetura Maresia. Portanto, o Capítulo seguinte descreve detalhadamente o ambiente de testes, as *workloads* e a metodologia além de discutir os resultados obtidos em todas as avaliações.

5 TESTES E RESULTADOS

A proposta do trabalho é de avaliar o desempenho do protótipo do modelo Maresia e fazer uma comparação com o Hadoop. Adicionalmente, deve-se identificar gargalos para que estes sejam otimizados no desenvolvimento de um novo *framework* que deverá utilizar esta arquitetura. Para isso, foram desenvolvidas algumas funcionalidades, cujo objetivo era de fazer com que o protótipo se assemelhasse mais com o Hadoop, implementando processos que anteriormente eram simplificados.

Em sua versão original, o protótipo possuia apenas um fator parametrizável que tinha influência no fluxo de execução (o número de tarefas de *Reduce*). Com os elementos desenvolvidos, há agora mais fatores interessantes que são passíveis de parametrização. Com isso, pode-se realizar uma avaliação de desempenho seguindo projetos experimentais, definindo fatores e níveis de variação para realizar este tipo de análise.

Este Capítulo tem como objetivo descrever a metodologia de experimentação utilizada, o ambiente de testes, as *workloads*, os resultados obtidos e uma discussão a respeito deles.

5.1 Ambiente de Testes

A avaliação dos *frameworks* foi realizada utilizando recursos da *grid* computacional francesa com a qual o grupo de pesquisa tem parceria (Grid'5000 (GRID'5000, 2014)). Este ambiente é separado em Sites, distribuídos por toda a França, e cada um deles contém um conjunto de um ou mais *clusters*.

O Site escolhido para testes foi Nancy, devido ao grande número de nós no *cluster* Graphene (144 Máquinas). A configuração de suas máquinas pode ser visualizada na Tabela 5.1.

Componente	**Modelo**
Processador	Intel Xeon X3440 2,53 Ghz 1 CPU x 4 Cores
Memória	16 GB SDRAM
Rede	Infiniband 20G (Mellanox MHGH29-XTC)
Armazenamento	SATA II 320GB

Tabela 5.1: Configuração de máquinas do *cluster* Graphene.

Para a escolha do *cluster* a ser utilizado, foi considerada a capacidade de processamento e paralelização dos nós. A *workload* gerada possui 8 gigabytes, permitindo a realização de experimentos com até 128 *chunks* de 64 megabytes. Portanto, pode-se sobrecarregar até 32 máquinas com 4 cores (distribuindo 4 *chunks* por máquina, fazendo

com que cada core tenha um para processamento) e como o ambiente escolhido possui 144 nós, não haverá problemas de quantidade de recursos. Além disso, para os experimentos menores, não é desejado que haja problemas de disponibilidade. Para tentar evitar isso, o *cluster* selecionado é um dos maiores que a Grid'5000 possui.

Os cenários utilizados para testes foram definidos com 3, 6 e 32 máquinas de processamento. O objetivo dos testes nestes cenários é apresentado na próxima Seção.

5.2 Metodologia

O protótipo será analisado realizando-se projetos experimentais, cujo objetivo é obter o máximo de informações com o mínimo de experimentos (JAIN, 1991). Com as modificações implementadas há três fatores interessantes a serem variados, os quais foram tomados como primários.

Será realizada uma experimentação $2^k r$, que é comumente utilizada na triagem de uma grande quantidade de fatores, para identificar aqueles que são mais impactantes. Ainda assim será utilizada esta metodologia pois ela também permite avaliar a contribuição de cada fator e dos erros no resultado.

Há três conjuntos de elementos que serão avaliados. Inicia-se por uma avaliação comportamental do protótipo, com o objetivo de identificar anomalias nos resultados, identificar gargalos e fazer uma avaliação de *speedup* (BAER, 2009). Posteriormente, realiza-se um comparativo entre o Hadoop e o protótipo, avaliando as diferenças nos resultados e os fatores mais impactantes de cada um. Faz-se ainda, um teste de escalabilidade simples, onde executa-se o protótipo sobre um grande número de nós para garantir seu funcionamento e avaliar seu comportamento.

Definiu-se dois conjuntos de observações de experimentos. No primeiro, 6 observações serão utilizadas pois estes são testes preliminares que objetivam ter uma noção do comportamento do protótipo e fazer uma comparação inicial com o Hadoop. Estabeleceu-se 30 replicações para o segundo conjunto, número largamente utilizado em experimentações, inicialmente aplicado em (STUDENT, 1908), que concluiu que com este número o valor médio da amostragem aproxima-se do valor real. O objetivo deste segundo conjunto é obter-se uma alta confiabilidade nos resultados.

Serão utilizadas 3 máquinas de processamento para a realização do primeiro conjunto de testes, pois não são desejados problemas em alocação de recursos nestes experimentos. Este é o número mínimo com que os *frameworks* conseguem executar. É importante salientar que devido à utilização de um nó *Master* pelo Hadoop, serão alocados 4 nós para haver 3 para processamento.

A segunda configuração será realizada sobre os mesmos recursos de ambiente, porém com duas vezes o tamanho da *workload*. Esta objetiva analisar um ambiente sobrecarregado, além de ser o primeiro passo para uma posterior avaliação de *speedup*.

O próximo cenário deve ter o dobro de processamento com a mesma *workload*. Portanto, 6 máquinas de processamento farão parte do ambiente. Com os resultados deste conjunto de testes, deve-se realizar a avaliação comportamental do protótipo e uma avaliação de *speedup*, em relação ao experimento anterior.

Finalmente, deve ser feito um teste de escalabilidade. Este experimento deve ser realizado apenas com o protótipo, sobre um ambiente com 32 máquinas, verificando o comportamento do protótipo em um ambiente com um acentuado número de nós. O número de *replicações*[1] a serem executadas será definido de acordo com o comportamento obser-

[1] A repetição de todos ou alguns experimentos é denominada *replicação*. Por exemplo, se todos os

vado.

Portanto, inicialmente tem-se experimentos com poucas repetições para avaliar preliminarmente o comportamento do sistema, verificando se não há nenhuma anomalia comportamental nos testes decorrente de algum *bug*, ou algum ponto na implementação que não está otimizado. Posteriormente tem-se uma avaliação rigorosa com um grande número de repetições, para dar confiança aos resultados obtidos nos estudos preliminares. Faz-se ainda uma verificação de como o protótipo se comporta sendo executado em um ambiente maior. Caso haja algum fator que necessite de uma avaliação mais detalhada, pode-se definir novos níveis e realizar novas experimentações a fim de fazer uma análise mais aprofundada do impacto deste fator nos resultados.

5.2.1 Aplicação de Teste

Como aplicação de testes, optou-se por utilizar o *WordCount*. Essa aplicação, faz parte do MRBS (abreviação de *MapReduce Dependability and Performance Benchmarking*) (SANGROYA; SERRANO; BOUCHENAK, 2012).

Adicionalmente, essa aplicação possui implementações tanto para o Hadoop quanto para o protótipo. Isso diminui o trabalho de implementação, visto que o foco do trabalho é na avaliação do desempenho e não no desenvolvimento de aplicações.

5.2.2 Definição de *Workload*

Na fase inicial do trabalho, já havia sido definido que a entrada utilizada para testes deveria se aproximar do que é utilizado em *jobs* reais do MapReduce. Isso decorre da frequente utilização de *workloads* geradas sintéticamente em avaliações. O motivo desta preocupação é que essas *workloads* sintéticas, geralmente são geradas utilizando uma semente (um arquivo com um determinado número de palavras) e concatenando-a até gerar o tamanho desejado de entrada. Esse procedimento pode prejudicar os resultados, dependendo do que está se avaliando.

Para fins de exemplificação, supõe-se um arquivo semente sendo utilizado para a geração da entrada, o qual possui 170 palavras diferentes em Inglês. A média de letras em uma palavra da língua Inglesa é de 4,5 letras (PIERCE, 2012), mas para facilidade de cálculo considera-se 5 letras. Tendo em vista que o arquivo semente (que além de palavras, contém espaços) tenha 1024 bytes, para gerar uma *workload* de 1 gigabyte, seria necessário concatenar o arquivo aproximadamente 1 048 576 vezes. No entanto, a execução da fase de *Combine* (que no caso do *WordCount*, diminui o tamanho das listas de pares intermediários para o número de palavras diferentes que um *chunk* possui), resultaria em no máximo 170 entradas na lista resultante da tarefa. Como a lista de pares intermediários é constituida de pares <*chave, valor*>, sendo *chave* a palavra e *valor* um inteiro de 4 bytes, resultaria em um total de 850 (5 letras * 170 palavras) + 680 (4 bytes de tamanho de int * 170 palavras) = 1530 bytes a serem transmitidos por tarefa.

Agora será exemplificado um caso extremo, onde supõe-se que a *workload* utilizada tenha 1 gigabyte de tamanho e 1 000 000 palavras diferentes (número estimado de palavras na língua incluindo termos científicos (MCCRUM; MACNEIL; CARN, 2012)). Após o *Combine* tem-se 5 000 000 (5 letras * 1 000 000 palavras) + 4 000 000 (4 bytes de tamanho de int * 1 000 000 palavras) = 9 000 000 bytes de pares intermediários a serem transmitidos. Note-se que este valor é 5882 vezes maior que o caso com a entrada gerada por um arquivo semente, e que este fato impacta diretamente no tempo de transmissão de

experimentos em um estudo são repetidos três vezes, é dito que o estudo teve três replicações (JAIN, 1991).

pares intermediários, no tempo de *Reduce* e consequentemente no tempo total do *job*.

Sendo assim a procedência dos dados de entrada pode impactar de uma forma significativa nos resultados. Portanto, para garantir que os resultados não seriam prejudicados pela *workload*, utilizamos as entradas referenciadas em (SANGROYA; SERRANO; BOUCHENAK, 2012), que podem ser encontradas em (WIKIPEDIA DUMPS, 2014). Estes dados são *dumps* das páginas da Wikipedia e estavam em formato XML. Foi necessário realizar um pré-processamento para retirar estruturas advindas da formatação, que poderiam interferir nos resultados, para a utilização nos experimentos.

Sobre o tamanho das entradas, esse parâmetro será justificado para cada teste, considerando o objetivo do mesmo.

5.3 Experimentos e resultados

Esta Seção apresenta os experimentos realizados, os resultados obtidos e uma discussão sobre eles. Como os objetivos da análise dos sistemas são avaliar o comportamento do protótipo, verificando seus gargalos, além de realizar uma comparação de seu desempenho com o Hadoop, optou-se por organizar a Seção inicialmente dissertando sobre a análise do comportamento do protótipo, em seguida, fazendo a comparação dos dois *frameworks* e finalmente realizando um teste de escalabilidade.

Observe que os resultados estão sumarizados da forma que melhor se encontrou para sua explicação e não na ordem em que foram executados. Como citado anteriormente, inicialmente realizou-se um conjunto de experimentos com 6 observações em cada configuração, com o objetivo de ter uma idéia do comportamento do sistema. Feito isso, foram disparados os experimentos com 30 repetições, em paralelo com o desenvolvimento de ferramentas que objetivavam a extração de informações dos arquivos gerados pelo sistema e a análise dos resultados das experimentações já realizadas.

Para melhor organização de tabelas, relacionaremos os fatores com um identificador. A Tabela 5.2 mostra a relação entre eles. Note-se que eles podem ser combinados, por exemplo, AB identifica a interação entre A e B.

Fator	**Identificador**
Slots de *Map*	A
Slots de *Reduce*	B
Número de tarefas de *Reduce*	C

Tabela 5.2: Relação de fatores e identificadores utilizados nas tabelas deste Capítulo.

Para uma melhor organização, dividiu-se a discussão dos resultados por configuração. A Seção 5.3.1 discute os resultados da análise comportamental do protótipo em sua primeira configuração, a Seção 5.3.2 os da segunda e a Seção 5.3.3 os da terceira, além de realizar uma análise de *speedup* do protótipo. A Seção 5.3.4 disserta sobre os resultados da análise comparativa com todas as configurações. Finalmente, os testes de escalabilidade do protótipo são apresentados em 5.3.5.

5.3.1 Análise comportamental - Configuração I

Os atributos desta configuração podem ser visualizados na Tabela 5.3. Os fatores disponíveis para variação e seus níveis são exibidos na Tabela 5.4.

Em relação ao ambiente, foram utilizados poucos nós, facilitando tanto a gerência

Atributo	Valor
Número de Máquinas	3
Tamanho da Entrada	768 MB
Tamanho de *Chunk*	64 MB
Número de *Chunks*	12

Tabela 5.3: Ambiente de testes do primeiro cenário de experimentação.

Fator	Níveis
Map Slots	1 e CoreNo (4)
Reduce Slots	1 e CoreNo (4)
Reduce Tasks	1 e 0,95*(nodes*max(ReduceSlots))

Tabela 5.4: Fatores e níveis do primeiro cenário de experimentação.

quanto a alocação. A *workload* foi dimensionada de forma que cada máquina fosse responsável por 4 *chunks*, que é o número de cores que as máquinas possuem.

Dois dos fatores escolhidos para utilização foram aqueles os quais o protótipo foi modificado para suportar. O fator *Map Slots* é referente ao número máximo de tarefas de *Map* que o sistema executa em paralelo, *Reduce Slots* é análogo ao anterior para tarefas de *Reduce* e *Reduce Tasks* é o número de reduções que serão executadas no *job*.

Sobre os níveis, primeiramente há apenas um *slot* de mapeamento e redução. Estes configuram os valores mínimos que se pode definir para estes parâmetros. O próximo nível foi definido com base no número de cores existentes nas máquinas, pois este será o número máximo de tarefas que é possível processar em paralelo. Portanto, neste nível há quatro *slots* definidos, tanto de *Map* quanto de *Reduce*.

O número de reduções foi definido da seguinte forma: primeiramente, tem-se o mínimo possível (apenas uma tarefa), o próximo nível é definido com base em informações da Wiki do Hadoop, que tem considerações sobre como otimizar este parâmetro do sistema (WIKI, 2014). Este define a quantidade ótima de tarefas de *Reduce* de acordo com a fórmula apresentada como segundo nível deste parâmetro, na Tabela 5.4.

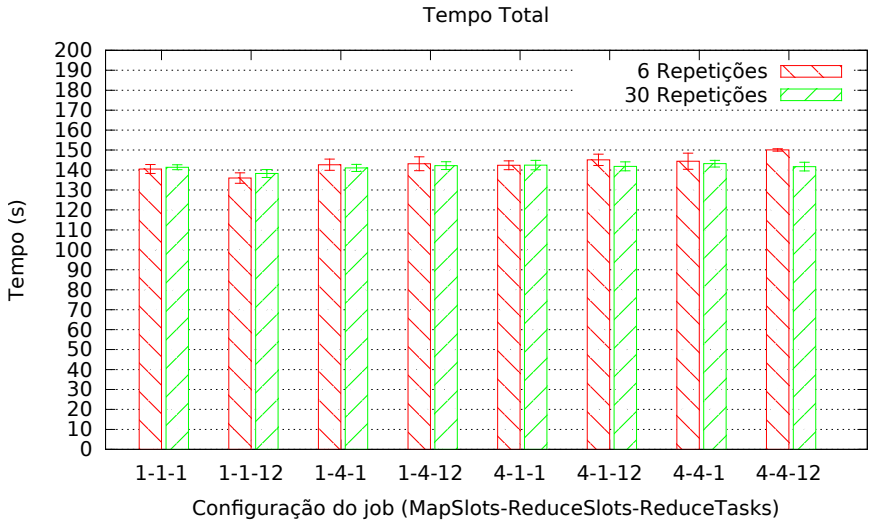

Figura 5.1: Duração total do *job* MapReduce, para 6 e 30 repetições, processando 768 megabytes em 3 máquinas.

A média do tempo total da execução deste experimento, com confiabilidade de 90% para 6 repetições e 95% para 30 repetições, pode ser visualizada na Figura 5.1. A Tabela 5.5 exibe a *alocação de variação*[2] de cada fator e de suas combinações.

Este não foi um resultado esperado, visto que a parcela de impacto atribuída a erros é muito alta e que os resultados praticamente não variam seu tempo de execução à medida que modifica-se os níveis dos fatores. Com 30 repetições, 93,29% da variação não pode ser explicada, e foi atribuída a erros. Optou-se então, por realizar uma análise aprofundada sobre cada parte da computação para identificar o que exatamente estava agregando uma taxa de erros desta magnitude no resultado final.

Fator	Alocação de Variação 6 Repetições	Alocação de Variação 30 Repetições
A	21,6531830626%	2,1876971793%
B	14,8838335134%	0,9687000365%
C	1,0560697121%	0,9306233062%
AB	0,3043919191%	0,5041002614%
AC	8,6843085518%	0,0007285998%
BC	3,5760093928%	0,6574465997%
ABC	0,2441483795%	1,4518265831%
Erros	49,5980554686%	93,2988774339%

Tabela 5.5: Alocações de variação geradas com um experimento 2^kr, utilizando o tempo total do *job* como variável de resposta.

Tomamos agora como *variável de resposta*[3], apenas a duração da computação da fase de *Map*. Esta fase engloba o processamento de todas as tarefas de mapeamento, além da execução do *Combine*. A Figura 5.2 exibe os resultados obtidos neste procedimento.

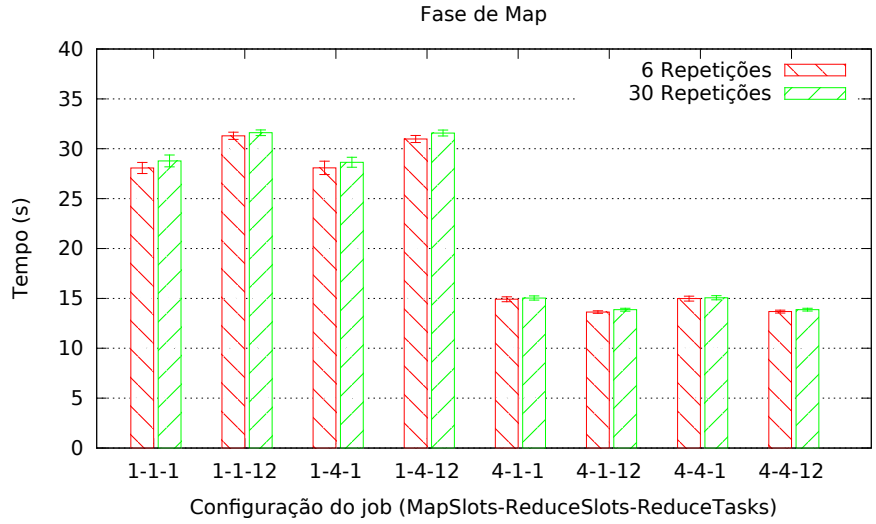

Figura 5.2: Duração da fase de *Map* processando 768 megabytes em 3 máquinas.

O comportamento deste passo da computação respondeu muito bem à alteração de níveis. Os *slots* de mapeamento, de acordo com a tabela 5.6, foram responsáveis por 97,60% da variação do resultado no conjunto de experimentos com 6 repetições e por 97,89% no com 30 repetições.

[2] A porcentagem de responsabilidade que um fator tem sobre o resultado final.
[3] Medida de performance do sistema sendo utilizada.

Este fator é responsável por esta parcela de variação dos resultados pois trata-se do número de *threads* que estão executando paralelamente tarefas de *Map*. Como atribuiu-se quatro *chunks* a cada *peer*, cada um responsabiliza-se por executar quatro tarefas. Quando modificamos este fator do primeiro para o segundo nível, é utilizado o paralelismo local das máquinas, pois estas possuem quatro cores, fazendo com que haja um ganho significativo de desempenho.

Fator	Alocação de Variação 6 Repetições	Alocação de Variação 30 Repetições
A	97,6052819974%	97,8967588455%
B	0,0009646213%	0,000480062%
C	0,3239007675%	0,2850139006%
AB	0,0045700045%	0,0009473434%
AC	1,971016971%	1,6384383259%
BC	0,0032851308%	0,0001698942%
ABC	0,0027092819%	0,0004266286%
Erros	0,0882712256%	0,1777649997%

Tabela 5.6: Alocações de variação geradas com um experimento 2^kr, utilizando a fase de *Map* como variável de resposta.

Além do fator citado, nota-se que há uma pequena parcela de variação dos resultados atribuída à interação dos *slots* de mapeamento com o número de tarefas de *Reduce*. Cada tarefa de *Map* gerencia e gera como saida *n* listas, onde *n* é igual ao número de tarefas de *Reduce* e por isso há este efeito da interação destes dois fatores. Cada lista de saída do mapeamento ainda deve passar pelo *Combine*, que também influenciará no resultado visto que ele trabalha gerando uma árvore binária, onde a organização dos pares intermediários nas listas de entrada afeta o seu desempenho.

Figura 5.3: Duração da fase de *Reduce* processando 768 megabytes em 3 máquinas.

Serão avaliados os resultados utilizando agora a duração da fase de *Reduce* como variável de resposta. Esta é mensurada a partir do início da transferência de dados intermediários até o final do *job*, incluindo o processamento de tarefas de redução e a exibição das saidas ou gravação destas em arquivo. A Figura 5.3 exibe os resultados obtidos e a Tabela 5.7, a alocação de variação de cada fator e suas interações.

Fator	Alocação de Variação 6 Repetições	Alocação de Variação 30 Repetições
A	10,7510488029%	0,0020055084%
B	16,5611418245%	0,980662376%
C	2,6694748761%	0,2289109176%
AB	0,3733675794%	0,533043988%
AC	11,0529470323%	0,0546191269%
BC	3,844026409%	0,6491698208%
ABC	0,2684447603%	1,5414065775%
Erros	54,4795487155%	96,010181685%

Tabela 5.7: Alocações de variação geradas com um experimento 2^kr, utilizando a fase de *Reduce* como variável de resposta.

Os resultados do conjunto de experimentos com 6 repetições não foram suficientes para obter-se uma média de resultados estável. Verificando o resultado com 30 observações, nota-se que este passo do processamento engloba aquilo que está fazendo com que haja uma altíssima atribuição de responsabilidade a erros na avaliação do *job*.

Para identificar com precisão o que exatamente esta prejudicando o protótipo, será realizada uma análise ainda mais detalhada. Esta consiste em avaliar separadamente o processamento de tarefas de *Reduce* e a transferência de pares intermediários.

Inicialmente, descartou-se o tempo de transferência, utilizando como variável de resposta apenas a execução de tarefas de redução. A Figura 5.4 exibe os resultados obtidos, e a Tabela 5.8 a alocação de variação de cada fator nesses resultados.

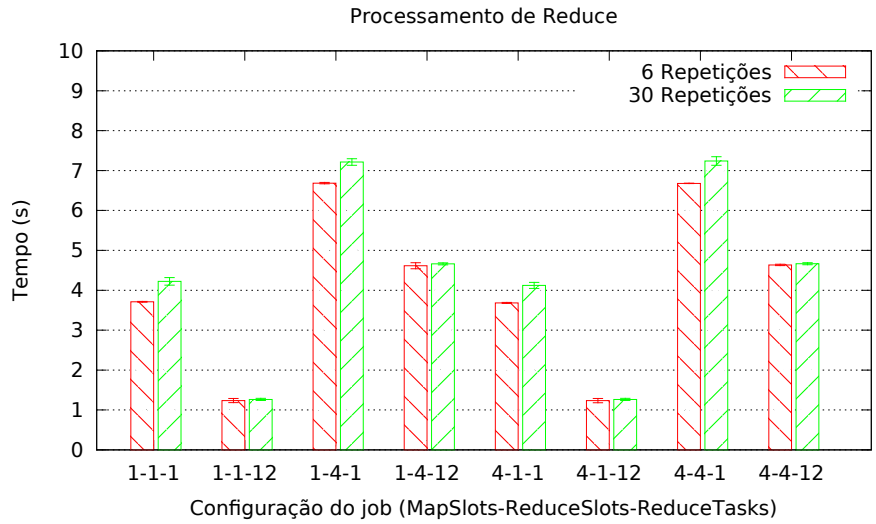

Figura 5.4: Duração do processamento da fase de *Reduce* (desconsiderando a transferência de dados intermediários), processando 768 megabytes em 3 máquinas.

Observa-se que considerando apenas o processamento, o resultado varia bastante com a modificação de níveis. Tanto nos resultados com 6 observações quanto nos com 30, é apresentado uma maior responsabilidade do fator *slots* de *Reduce*, em segundo lugar fica o número de tarefas de redução. A quantidade atribuída a erros é próxima de 0 e por exclusão, pode-se concluir que o processo que está adicionando uma responsabilidade significativa dos erros nos resultados é a transferência de dados intermediários.

O fator número de tarefas de *Reduce* impacta de forma significativa nos resultados, pois ele é responsável pelo tamanho da tarefa de *Reduce*. Quanto maior este número,

menores serão as tarefas, pois a quantidade de dados que cada uma deve processar será menor.

Fator	Alocação de Variação 6 Repetições	Alocação de Variação 30 Repetições
A	0,0007042341%	0,0017783678%
B	67,3600877522%	58,597338773%
C	32,4323420866%	40,4752982684%
AB	3,52069436626353E-005%	0,0058297618%
AC	0,0003316699%	0,0023873234%
BC	0,1983783474%	0,1628119729%
ABC	0,0005784746%	0,005374405%
Erros	0,0075422282%	0,7491811278%

Tabela 5.8: Alocações de variação geradas com um experimento 2^kr, utilizando a duração do processamento da fase de *Reduce* como variável de resposta (desconsiderando a transferência de dados intermediários).

Os *slots* de redução são responsáveis pela maior variabilidade dos resultados. Porém analisando isso em relação ao gráfico de resultados nota-se que, com o aumento dos *slots*, tem-se um prolongamento na duração do processamento. Isso é contra-intuitivo visto que se há mais *threads* processando e as máquinas possuem recursos para executá-las em paralelo, deveria haver um ganho no desempenho (como aquele apresentado na fase de *Map*).

Realizando uma verificação minuciosa dentro do protótipo, constatou-se que duas ações contribuem para este comportamento. A primeira delas é a metodologia de criação de *threads*, a qual cria sempre aquelas relacionadas a tarefas de mais alto identificador primeiro. Logo, se há quatro *slots* e uma tarefa, primeiramente são criadas três *threads* que não têm tarefas relacionadas para depois criar a tarefa que efetivamente tem dados. A segunda ação é a utilização de um *delay* de 1 segundo após a criação de cada *thread*, utilizado para definir algumas variáveis de estado para a próxima a ser executada.

Para exemplificar essa questão, serão utilizados dois conjuntos de observações. Observa-se que em ambos não há variação dos *slots* de mapeamento, pois esse fator utilizando o processamento da fase de *Reduce* como variável de resposta é responsável por uma porcentagem desprezível nos resultados (próxima de 0).

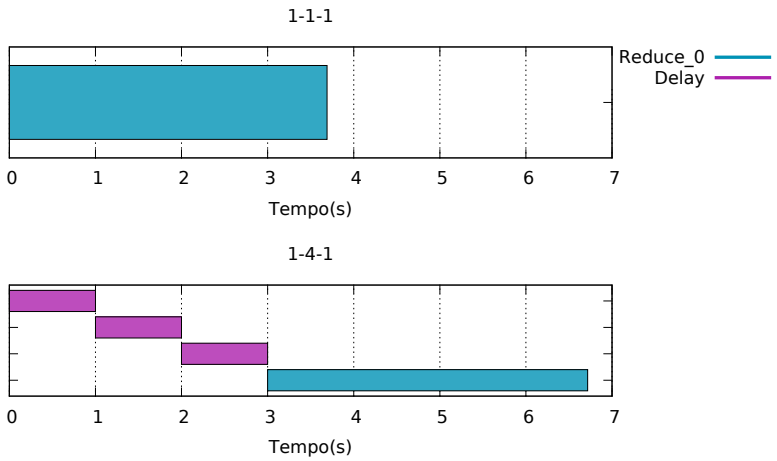

Figura 5.5: Diagrama de Gantt referente à variação de *slots* com 1 tarefa de *Reduce*.

A Figura 5.5 exibe o efeito da variação do fator discutido, quando há apenas uma tarefa de *Reduce* definida. Esperava-se que o tempo permanecesse o mesmo nessas configurações, pois tem-se apenas uma tarefa alocada em ambas. No entanto, na Figura pode ser visualizado claramente a influência dos efeitos citados, ou seja, a criação de fluxos começa do maior identificador (como há apenas uma tarefa, não há nenhum processamento nas três primeiras *threads*) e só então é criada aquela que é responsável pela execução. Note ainda, que a duração da tarefa *Reduce_0* é praticamente igual em ambos os casos (3,69s para a tarefa do gráfico superior e 3,71s do gráfico inferior).

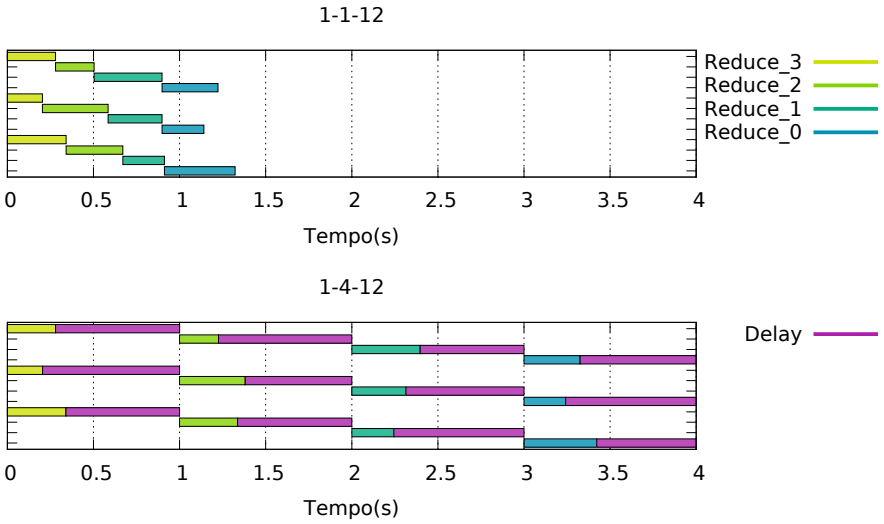

Figura 5.6: Diagrama de Gantt referente à variação de *slots* com 12 tarefas de *Reduce*.

Agora analisando o cenário com mais de uma tarefa, a Figura 5.6 apresenta o diagrama de Gannt gerado para esse conjunto de observações. Note que este diagrama é um pouco diferente do anterior, ele exibe os resultados de todas as máquinas (no anterior havia apenas uma máquina pois havia apenas uma tarefa no ambiente inteiro), onde cada uma delas executa um conjunto de quatro tarefas.

Com essas configurações, esperava-se um ganho de desempenho ao utilizar um número maior de *slots*. Todavia, não é isso o que se observa nos resultados, e o principal responsável por essa degradação do desempenho é o *delay* de criação de *threads* (associado com o tamanho das tarefas de *Reduce*, que são bem pequenas). Note ainda que neste caso, a ordem de criação de *threads* não degrada o desempenho, visto que todas aquelas que forem iniciadas terão uma tarefa para processar.

Portanto, o desempenho do processamento da fase de *Reduce* é prejudicado por algumas decisões de implementação que se mostraram ineficientes. Para o desenvolvimento do novo *framework*, é necessário eliminar os elementos que contribuiram para essa degradação de performance.

Agora será investigado o que foi constatado como o gargalo do sistema, a transferência de dados intermediários. A Figura 5.7 exibe os resultados obtidos utilizando esta como variável de resposta para análise e a Tabela 5.9, a alocação de variação de cada fator.

Utilizando os resultados observados com 30 repetições (pois são mais confiáveis), observou-se que 99,01% da variação neste processo não pode ser explicado pela variação dos fatores e é atribuído a erros. Como a transferência destes dados intermediários é o processo mais oneroso de todo o *job*, uma alta taxa de erros nela influi diretamente na taxa de erros do *job*. Para identificar as causas da performance apresentada e da variação

atribuída a erros, o código referente a este processo foi estudado, onde foram identificadas diversas decisões de implementação que resultaram no que foi observado.

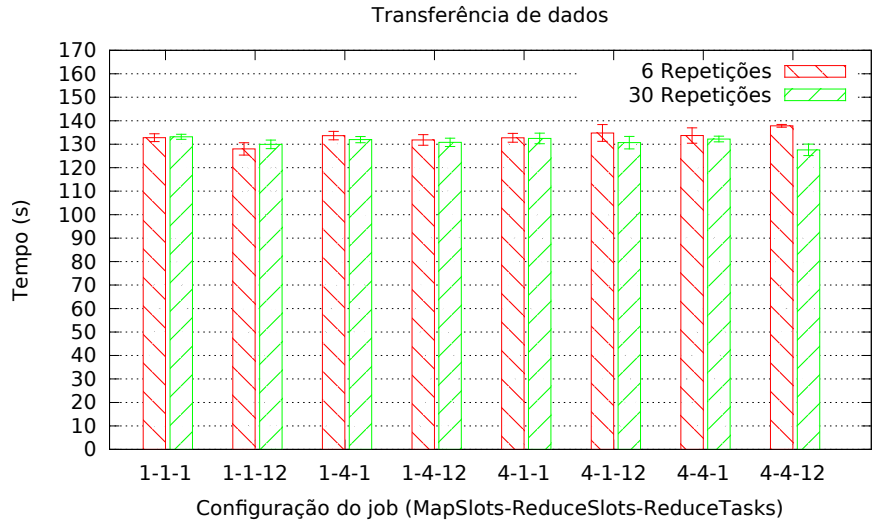

Figura 5.7: Duração da transferência de dados intermediários, processando 768 megabytes em 3 máquinas.

É importante salientar que o ambiente de testes é composto por uma rede Infiniband 20G, a qual pode chegar a uma taxa de transferência de 20Gb/s (MELLANOX, 2014). Portanto exclui-se a possibilidade de a ineficiência observada ser atribuída à capacidade da rede.

Para entender o que contribui para a lentidão na transferência de dados, primeiramente precisa-se entender o *workflow* deste processo. O nó transmissor finaliza uma tarefa de *Map*, gerando *n* listas de pares intermediários (onde *n* é o número de *reduces*). O identificador utilizado para elas é o mesmo utilizado para as tarefas de redução, e elas são enfileiradas em ordem crescente de identificador para o envio pela *thread* de transmissão de dados.

Fator	**Alocação de Variação 6 Repetições**	**Alocação de Variação 30 Repetições**
A	12,7786414397%	0,002108551%
B	5,4854210134%	0,10635672%
C	0,2165806885%	0,5077669985%
AB	0,4825107421%	0,0969120279%
AC	12,8815721563%	0,0020477382%
BC	1,6881398553%	2,92377979913038E-005%
ABC	0,5549409135%	0,2714839871%
Erros	65,9121931911%	99,0132947395%

Tabela 5.9: Alocações de variação geradas com um experimento 2^kr, utilizando a duração da transferência de dados intermediários como variável de resposta.

Para cada lista a ser transmitida é aberta uma conexão TCP com o receptor, o qual cria uma *thread* para a recepção destes dados. Ao enviar uma lista, para cada elemento <*chave, valor*> é enviada uma mensagem TCP, e para cada mensagem transmitida, o remetente aguarda um ACK no nível de aplicação para prosseguir com o envio. Ao receber uma mensagem o receptor verifica a integridade dos dados, obtém um semáforo para

a gravação dos dados em arquivo, salva-os, envia um ACK e libera o semáforo obtido anteriormente.

Um dos responsáveis pela performance apresentada é a quantidade de aberturas e fechamentos de conexões TCP. O motivo disso é que cada tarefa de *Map* (há 4 tarefas de *Map* por *peer* no conjunto de testes em questão) gera *n* listas, onde *n* é o número de *reduces*. Para configurações onde tem-se apenas 1 tarefa de *Reduce*, isso não prejudica muito o desempenho, pois para cada *chunk* haverá apenas 1 conexão criada e destruída. No entanto, quando passa-se para o segundo nível deste fator, há um total de 48 conexões criadas e destruidas (12 por *chunk*), o que contribui para a ineficiência deste processo.

Outro fator que contribui significativamente é a forma como os dados são enviados, com cada par intermediário em uma mensagem. Alguns dezenas de milhares de pares intermediários compõem uma lista dessas (dependente de quantas tarefas de *Reduce* configuradas), sendo assim, além da ineficiência há o *overhead* de cabeçalhos de mensagem (que no caso de TCP + IPv4, são 40 bytes) multiplicado dezenas de milhares de vezes. Além disso, o ACK aguardado no nível de aplicação dobra a quantidade de mensagens trafegadas. Todo esse processo pode ser otimizado fazendo pacotes com mais de um par <*chave, valor*> para envio, tendo em vista que podem ser enviados até 1460 bytes de conteúdo (1500 bytes é o tamanho máximo de pacote que pode-se transmitir, tirando os 40 bytes de headers IPv4 + TCP), o que diminuiria consideravelmente o número de mensagens. Ainda, pode-se dimensionar o sistema de forma que seja utilizada a confiabilidade provida pelo protocolo de transporte, fazendo com que não seja necessária a utilização de ACKs no nível de aplicação.

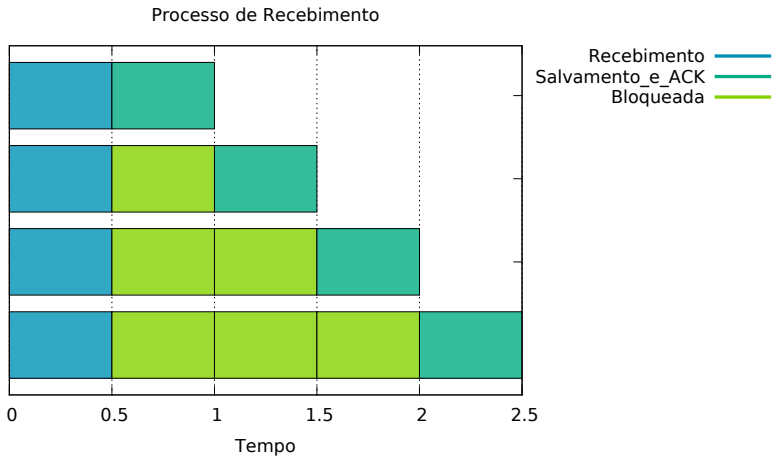

Figura 5.8: Simulação do processo de recebimento de dados.

Um grande contribuinte para o desempenho resultante é o processo de recebimento de dados intermediários. A presença do semáforo combinada com o ACK logo após a gravação do conteúdo recebido contribui para a lentidão. Como há várias *threads* recebendo dados, frequentemente referentes a mesma tarefa, o tempo para adquirir o semáforo pode acabar propagando um atraso para o envio de dados de todas as *threads*.

A Figura 5.8 auxilia no entendimento do processo descrito. Note-se que a unidade utilizada na Figura é uma unidade de tempo genérica, pois é apenas uma simulação de como isso ocorre e tem-se quatro *threads* recebendo mensagens referentes a mesma tarefa de *Reduce*. Este é um caso extremo, onde recebeu-se quatro mensagens referentes a mesma tarefa de redução. Todas são recebidas simultâneamente, mas cada uma será gravada

separadamente em disco pois há exclusão mútua na escrita em arquivo de dados intermediários de uma mesma tarefa. É importante salientar que, enquanto o ACK não chegar no remetente, ele **não enviará novas mensagens**.

Conclui-se então, que a transmissão de dados intermediários é o grande gargalo do protótipo. Ela é responsável pela grande porcentagem de variação que não se consegue atribuir a um fator específico, e a baixa responsividade dos resultados quando varia-se os níveis dos fatores, ao utilizar a duração do *job* como variável de resposta. Mesmo na transferência de dados, esperava-se uma contribuição significativa do fator Tarefas de *Reduce*. Todavia, este fator teve uma pequena parcela de responsabilidade devido aos diversos pontos que não foram bem dimensionados, os quais tornaram a distribuição de dados extremamente ineficiente.

5.3.2 Análise Comportamental - Configuração II

Esta configuração difere da anterior apenas no tamanho de sua *workload*. As demais parametrizações do experimento, tanto em relação ao ambiente quanto aos fatores e níveis, são exatamente as mesmas.

O aumento da *workload* permite que na próxima Seção seja realizada uma análise de *speedup* utilizando o dobro da capacidade de processamento. Note-se que se fosse utilizado o *workload* anterior para uma avaliação deste tipo, haveria uma subutilização do ambiente pois cada *peer* é capaz de executar em paralelo o processamento de quatro *chunks*, no entanto, haveriam apenas dois.

A Figura A.1 apresenta o tempo total do *job* com essa configuração. Os resultados também não têm uma reatividade significativa com a variação dos fatores (Há uma diferença de cerca de 20 segundos entre o melhor e o pior caso). Observa-se um comportamento semelhante ao experimento anterior. De acordo com a Tabela B.1, as diferenças mais relevantes entre esses dois experimentos em relação à atribuição de responsabilidade de cada fator são:

- Diminuição da responsabilidade de erros;
- Aumento na variação do resultado decorrente do número de tarefas de *Reduce*. Verifica-se uma melhora no tempo total do *job* ao analisar seu gráfico.

O tempo total do *job* praticamente dobrou, o que é justo pois dobrou-se volume de dados a serem processados. Será adotada a mesma abordagem da análise anterior, observando diversas variáveis de resposta que compõem a execução, para entender melhor o que causou este resultado.

As médias e os intervalos de confiança da fase de *Map* são exibidos na Figura A.2 e a alocação de variação de cada fator é exibido na Tabela B.2. Como no experimento anterior, tem-se um grande impacto relacionado aos *slots*. Com um fluxo processando tarefas, tem-se o dobro do tempo em relação à configuração anterior, o que faz sentido pois há duas ao invés de uma tarefa de *Map* alocadas para cada *thread*. Com quatro *threads* há uma performance levemente melhor, pois como cada uma processa duas tarefas, torna o *overhead* de criação delas menos evidente em comparação com o experimento apresentado na Seção 5.3.1, onde era atribuída uma tarefa para cada.

A Figura A.3 apresenta os resultados obtidos da fase de *Reduce* desta configuração e as alocações de variação dos fatores podem ser verificadas na Tabela B.3. É apresentada uma responsabilidade de fatores semelhante ao total do *job*, pois esta fase engloba a maior parte

da duração dele. Para melhor analisar esta, avalia-se separadamente o processamento e a transmissão de dados, como foi feito na Seção anterior.

Inicialmente excluiu-se a transferência de dados, analisando apenas o processamento da redução. A Figura A.4 exibe os resultados obtidos. De acordo com a Tabela B.4, o número de tarefas de *Reduce* representou cerca de 73,81% da responsabilidade na variação dos resultados, um número 30% maior em relação ao experimento anterior. Como há mais dados para serem processados, é justo que ao dividi-los para a execução implique em um grande impacto no resultado.

Observou-se ainda que a variação do fator *slots* de *Reduce* teve um impacto menor no processamento (25,19% contra 58,59% da configuração anterior). Com o aumento dos dados de entrada, também cresce a quantidade de informação que precisa ser processada nesta fase, fazendo com que o *overhead* de criação das *threads* fique menos evidente. É importante destacar novamente que o impacto do aumento destes *slots* é **negativo**, ou seja, aumenta o tempo de processamento, e que os motivos disso foram explicados com detalhes na Seção anterior.

Utilizando a transferência de dados intermediários como variável de resposta, tem-se o resultado apresentado na Figura A.5. Note que este procedimento é o mais custoso de todo o *job*. Devido a isso, ele é responsável por definir o comportamento da execução inteira, como pode-se observar comparando a Tabela B.5 com a Tabela B.1.

É nele que observamos o crescimento da atribuição de responsabilidade do fator tarefas de *Reduce*, pois esse está diretamente associado à transmissão de dados. Além disso, verifica-se a presença significativa de erros, advindos das diversas características discutidas na Seção 5.3.1.

Sendo assim, essa configuração também apresenta uma baixa responsividade em relação à variação de níveis dos fatores. Novamente, a transmissão de dados intermediários introduziu uma quantidade significativa de erros no *job*, decorrente das características já discutidas. Dentre elas estão a criação de conexões TCP para o envio de **cada lista resultante do *Map***, a metodologia de envio destas, a qual transmite um par intermediário por mensagem para o receptor, e a presença de uma seção crítica no meio do processo de recepção destes dados, o que em alguns casos acaba por serializá-lo.

5.3.3 Análise Comportamental - Configuração III

Neste novo cenário de experimentação, utilizou-se o dobro do poder de processamento. Agora, temos não 3 mas 6 *peers* no ambiente de execução. A Tabela 5.10 sumariza esta configuração e na Tabela 5.11 são apresentados os fatores e níveis utilizados para variação. O objetivo deste experimento é, além de analisar o comportamento do protótipo, verificar seu *speedup* em relação à configuração anterior.

Atributo	Valor
Número de Máquinas	6
Tamanho da Entrada	1536 MB
Tamanho de *Chunk*	64 MB
Número de *Chunks*	24

Tabela 5.10: Ambiente de testes do terceiro cenário de experimentação.

Utilizou-se a mesma variação de níveis em *slots* de mapeamento e redução. Na definição do número de tarefas de *Reduce*, utilizou-se a mesma quantidade da configu-

Fator	Níveis
Map Slots	1 e CoreNo (4)
Reduce Slots	1 e CoreNo (4)
Reduce Tasks	1 e 12

Tabela 5.11: Fatores e níveis do terceiro cenário de experimentação.

ração anterior. O objetivo desta decisão foi de realizar a análise de *speedup* verificando apenas o efeito do aumento do número de máquinas.

Com o tempo total do *job* como variável de resposta o resultado apresenta, com a variação de níveis, uma responsividade semelhante aos obtidos anteriormente. Esta baixa variabilidade dos resultados pode ser verificada na Figura A.6, onde a diferença entre o melhor e o pior caso é de apenas 20s.

O *speedup* do tempo total do *job* em relação à configuração anterior foi em média de 1,58. Como dobrou-se o poder de processamento do ambiente, o ideal seria obter um valor em torno de 2, todavia, há diversos pontos que serão abordados em seguida que contribuem para que isso não ocorra.

Neste experimento foi atribuída uma responsabilidade ainda menor dos erros no resultado final (29,29% nos resultados com 30 observações da Tabela B.6). O número de tarefas de *Reduce* foi novamente responsável pela maior alocação de variação no resultado. Todavia, foi obtido um comportamento **diferente do anterior**, enquanto na Configuração II ao aumentar o número de tarefas de *Reduce* havia redução no tempo total, neste o protótipo apresenta um aumento. Para melhor verificar essa questão, será realizada o mesmo tipo de avaliação presente nas duas experimentações anteriores.

Tratando a fase de *Map* como variável de resposta não houveram surpresas, mostrando que este é um dos passos mais estáveis do protótipo. A Figura A.7 exibe os resultados obtidos da execução desta fase. O fator *slots* de mapeamento foi novamente responsável por quase a totalidade da variação (97,06% considerando 30 repetições de acordo com a Tabela B.7). A interação entre ele e o número de tarefas de *Reduce* apresentou uma alocação de variação de 2,13%, devido ao que já foi discutido.

No mapeamento, foi obtido 1,87 de *speedup* em comparação com a configuração anterior. Mesmo nesta que é a fase mais estável, não foi obtido o valor ideal pois nas configurações com 4 *slots*, há um *overhead* de tratamento destes que faz com que não obtenhamos exatamente metade do tempo de processamento anterior.

Analisando a fase de *Reduce*, é possível verificar a modificação no comportamento do protótipo citada anteriormente, conforme a Figura A.8. A Tabela B.8 apresenta a redução da alocação de variação de erros e o aumento da responsabilidade do número de tarefas de *Reduce* observados utilizando o tempo total do *job* como variável de resposta.

Em média, o protótipo apresentou 1,60 de *speedup* utilizando-se esta variável de resposta. Será verificado mais detalhadamente o que contribui para isso, realizando-se uma avaliação mais detalhada, onde separou-se o processamento da transmissão de dados, como realizado anteriormente.

Com o processamento do *Reduce* (excluindo-se a transmissão de dados intermediários da fase), é obtido um resultado tão estável quanto aquele da fase de *Map* (Figura A.9). Pode-se notar os *delays* de criação de *threads* já discutidos nos resultados, observando-se as configurações 1-1-1 e 1-4-1, por exemplo.

A Tabela B.9 mostra que a responsabilidade atribuída ao número de tarefas de *Reduce* é ainda maior que na experimentação anterior. O motivo disso é que com mais máquinas,

as tarefas acabam por ficar mais distribuídas. Portanto, a cada *peer* são alocadas menos tarefas (anteriormente eram alocadas 4 por *peer*, agora são apenas 2, considerando o nível com 12 tarefas), fazendo com que este processo seja mais rápido.

Com esta variável de resposta, obtivemos uma média de 1,15 de *speedup*. A causa disso é que, quando há apenas 1 tarefa de Reduce, não há como obter uma melhora no tempo de execução (tendo em vista que é realizado o mesmo processo nas duas configurações). Portanto, temos uma melhora apenas em configurações com o segundo nível de número de reduções. Na prática ela é pequena devido às questões não eficientes de implementação desta parte do *job*, como por exemplo, os *delays* de criação de *slots* de redução.

Foram analisadas todas as variáveis de resposta, salvo a transmissão de dados intermediários. Portanto, atribuiu-se a este passo a responsabilidade pela modificação observada no comportamento. Isso pode ser visualizado claramente na Figura A.10 e na Tabela B.10. A causa ainda precisa ser melhor investigada, mas ela tem relação com a quantidade de *threads* transmitindo dados simultâneamente.

O *speedup* obtido aqui foi de 1,62. Este valor sofre influência das diversas questões levantadas anteriormente sobre a transferência de pares intermediários e por isso apresenta um valor abaixo do ideal.

Sendo assim, concluiu-se que o grande gargalo do protótipo é indubitavelmente a transmissão de dados intermediários. A fase de *Map* está bem implementada e obtemos resultados próximos dos melhores possíveis com a variação dos níveis nela. O processamento do *Reduce* pode ser otimizado, no entanto, por ser responsável por tarefas extremamente curtas, não causou um maior impacto no resultado obtido.

Analisando o *speedup* obtido em geral, foi obtido apenas 1.58. Os processos realizados na fase de *Reduce* foram os principais responsáveis por este resultado. A implementação de forma não eficiente de alguns pontos do protótipo é o que mais contribuiu para isto.

Na implementação do novo *framework* para utilização real, deve-se melhorar substancialmente o processo de transferência de dados. Foram sugeridas diversas melhorias, por exemplo o envio de mais de um par intermediário por mensagem e a criação de apenas uma conexão para a transferência desses, que certamente contribuirão para o um melhor desempenho.

5.3.4 Análise Comparativa

Foram realizados experimentos no Hadoop seguindo a mesma metodologia utilizada com o protótipo. Inicialmente foram realizados conjuntos de testes com 6 observações, para ter-se uma idéia do comportamento do sistema e posteriormente foram disparados experimentos com 30 repetições. Com o objetivo de facilitar a apresentação de resultados, neste Capítulo serão utilizados apenas aqueles advindos de experimentações com o maior número de observações.

Utilizou-se as mesmas configurações de ambiente e os mesmos fatores e níveis, apresentados nas Seções passadas, referentes à análise comportamental. A Versão utilizada do Hadoop foi a 1.2.1. Como citado anteriormente, ele requer que sejam alocadas *n+1* máquinas para se ter *n* nós processando tarefas, pois o nó *Master* é responsável apenas pela gerência dos *jobs*.

A Figura 5.9 mostra a comparação entre os dois *frameworks*, no primeiro experimento. O Hadoop, em geral, se mostrou mais eficiente que o protótipo. Era esperado que ele apresentasse uma performance melhor, pois a análise comportamental mostrou diversos

pontos que não estão eficientes no protótipo e que precisam ser melhorados para que ele seja competitivo.

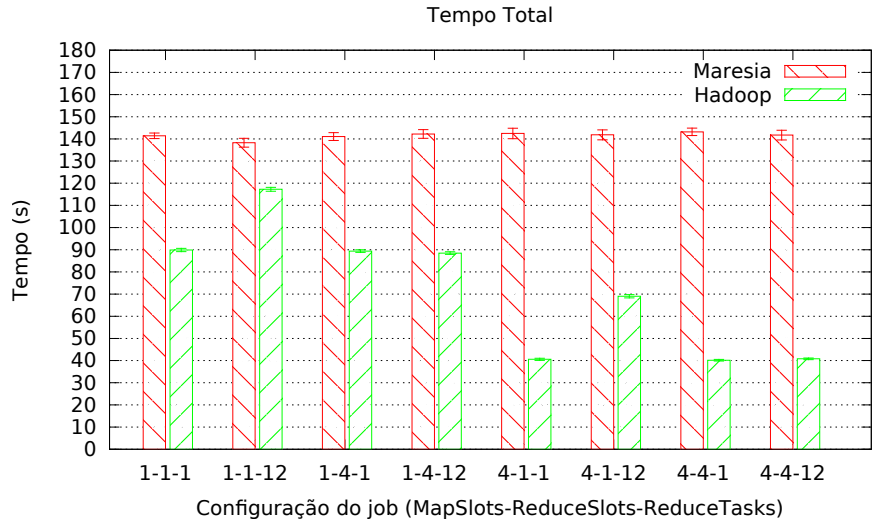

Figura 5.9: Comparação entre tempos de execução do protótipo e do Hadoop, processando 768 megabytes em 3 máquinas.

Avaliando os resultados exibidos nas Figuras A.11 e A.12, verifica-se que o Hadoop apresenta o mesmo comportamento para todas as configurações utilizadas (lembrando que foram utilizadas as mesmas configurações da análise comportamental). Devido a isso, será analisada apenas a primeira (mesmos ambiente e parâmetros utilizados no experimento da Seção 5.3.1), fazendo *links* as outras duas experimentações realizadas sempre que for pertinente.

Pode-se observar que, diferente do protótipo, o tempo total do *job* tem uma grande responsividade à variação de fatores no Hadoop. A Tabela 5.12 apresenta a comparação entre as alocações de variação dos dois *frameworks*, nas Tabelas B.11 e B.12 pode-se visualizar os resultados das outras configurações. Note-se que são utilizados os mesmos identificadores para os fatores, definidos no início deste Capítulo.

O Hadoop apresenta uma alocação de variação extremamente similar em todas as 3 configurações de experimentos. Tendo em vista que ele é um sistema já estável, é importante que ele apresente este tipo de comportamento. O protótipo apresentou resultados bem diferentes neste quesito e as causas disso já foram abordadas na análise comportamental.

Fator	**Maresia**	**Hadoop**
A	2,1876971793%	79,4550774945%
B	0,9687000365%	7,0391985331%
C	0,9306233062%	6,4758918973%
AB	0,5041002614%	0,0008679903%
AC	0,0007285998%	0,0142685095%
BC	0,6574465997%	6,6141117816%
ABC	1,4518265831%	0,0005054756%
Erros	93,2988774339%	0,400078318%

Tabela 5.12: Comparação de alocação de variação entre Hadoop e o protótipo, para o tempo total do *job*.

Tendo em vista a análise comportamental realizada anteriormente, é evidente que a primeira grande diferença é a responsabilidade atribuída a erros. O Hadoop comporta-se muito melhor pois não há gargalos que prejudicam tanto a sua execução. O fator *Map Slots* é o mais impactante. Ele tem uma grande responsabilidade na fase de *Map*, e ainda agiliza a fase de *Reduce*, pois quanto mais rápido o processamento dos mapeamentos, mais cedo pode-se enviar seus dados.

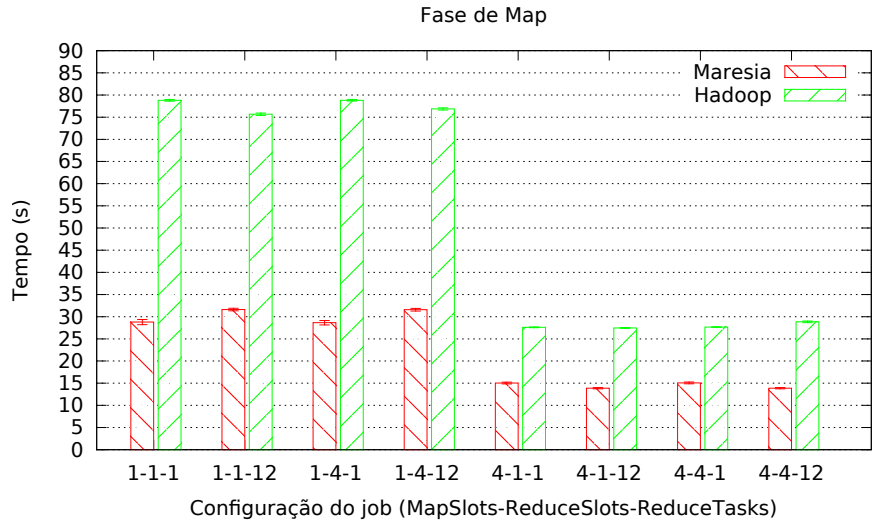

Figura 5.10: Comparação entre tempos de execução da fase de *Map* do protótipo e do Hadoop, processando 768 megabytes em 3 máquinas.

A Figura 5.10 mostra a comparação apenas da fase de Map de ambos os sistemas. Nas Figuras A.13 e A.14 são apresentados os resultados das duas outras configurações.

O protótipo foi o mais eficiente no processamento desta fase em todas as configurações. A linguagem de programação e o fato de ele tratar os dados intermediários em memória (enquanto o Hadoop executa gravação em disco) contribuem para isso. Não foram verificados quantitativamente o que contribui mais, pois não faz parte do escopo do trabalho uma comparação tão detalhada.

De acordo com a Tabela 5.13, há uma alocação de variação similar entre os dois sistemas. Entretanto, o protótipo tem uma pequena contribuição da interação entre os slots de mapeamento e o número de tarefas de reduce, citada na análise comportamental. As Tabelas B.13 e B.14 apresentam os dados das outras duas configurações.

Fator	**Maresia**	**Hadoop**
A	97,8967588455%	99,7889405949%
B	0,000480062%	0,0173270035%
C	0,2850139006%	0,041342168%
AB	0,0009473434%	0,0001325821%
AC	1,6384383259%	0,095598095%
BC	0,0001698942%	0,0161189873%
ABC	0,0004266286%	5,39631711330651E-005%
Erros	0,1777649997%	0,040486606%

Tabela 5.13: Comparação de alocação de variação entre Hadoop e o protótipo, para a fase de *Map*.

Sobre a fase de Reduce, a Figura 5.11 exibe os resultados obtidos processando 768 megabytes em 3 máquinas. As outras duas configurações são apresentadas nas Figuras A.15 e A.16.

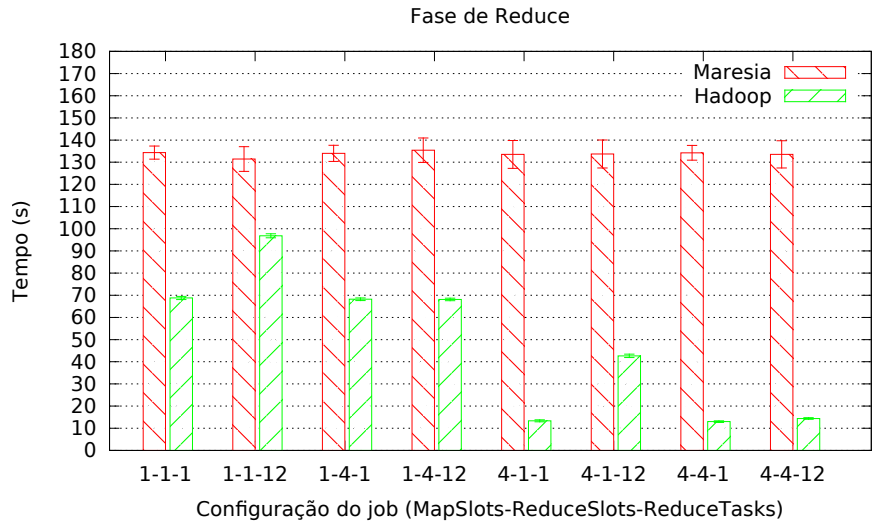

Figura 5.11: Comparação entre tempos de execução da fase de *Reduce* do protótipo e do Hadoop, processando 768 megabytes em 3 máquinas.

Nas três configurações, o Hadoop foi muito mais eficiente na execução do Reduce (apresentou o mesmo comportamento nas 3 configurações). Como as tarefas desta fase são pequenas, concluiu-se que sua eficiência é resultante de uma otimizada transferência de dados (que é o processo mais custoso deste passo de execução).

É importante salientar ainda que, na variação de número *slots* quando tem-se apenas 1 tarefa de *Reduce*, o Hadoop não apresenta variação no resultado. Esse ponto era esperado ao analisar o protótipo, todavia, verificou-se que esse não é o comportamento que ele apresenta (isso pode ser visualizado na Seção 5.3.1).

Notou-se ainda que o tempo da fase de redução aumentou consideravelmente quando tem-se 12 tarefas de Reduce sendo processadas em 1 slot, com apenas 1 thread de mapeamento. Nesta configuração, cada *peer* é responsável por 4 destas tarefas. É iniciada a primeira após o término do primeiro mapeamento, onde ela começa a obter os dados de entrada (pares intermediários de saída do mapeamento). A tarefa só pode processar esses dados após o término da fase de Map, quando garante que recebeu todos os pares necessários. Isso faz com que a duração desta fase seja prolongada (note que isso não ocorre quando temos a fase de Map processada com 4 slots, pois a espera para sua finalização é menor).

Procurou-se dividir a fase de Reduce entre transferência de pares intermediários e processamento, para realizar uma avaliação semelhante à comportamental. Todavia, os logs do Hadoop não forneciam dados suficientes. Seria interessante ter essa divisão pois possibilitaria uma análise comparativa mais detalhada da redução, utilizando essas variáveis de resposta.

Na tentativa de diminuir substancialmente a transmissão de dados intermediários para avaliação, executou-se também testes com uma *workload* gerada sinteticamente. Esta, tinha apenas 36 palavras de conteúdo, repetida diversas vezes. Com essa variabilidade de palavras, são gerados 36 pares intermediários para transferência, o que deveria ser

transmitido muito rápido se comparado a *workload* utilizada nos testes anteriores (que gerava dezenas de milhares).

Utilizou-se 3 máquinas de processamento e a *workload* gerada tinha 768 megabytes. A Figura 5.12 mostra o tempo total do *job*.

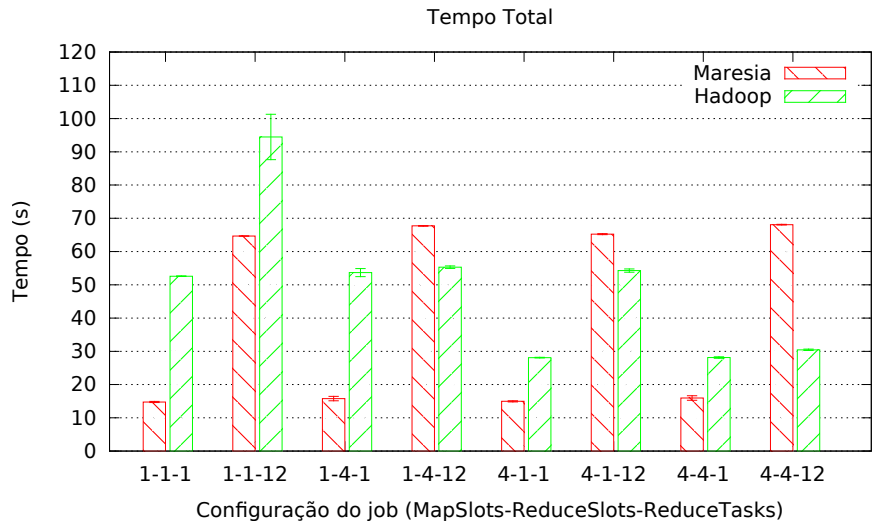

Figura 5.12: Comparação entre tempo total do *job* entre Hadoop e protótipo, processando uma *workload* sintética de 768 megabytes em 3 máquinas.

Nos casos onde tem-se apenas 1 tarefa de redução, o protótipo é mais rápido que o Hadoop. Quando este fator é definido para seu segundo nível, ele acaba sendo muito mais lento. Não identificou-se com precisão o que causa essa lentidão, pois claramente é relacionado à transmissão de pares intermediários e isso já foi constatado como um processo problemático. Os resultados da fase de mapeamento e redução analisados separadamente não acrescentam informações ao que foi observado. Eles podem ser visualizados nas Figuras A.17 e A.18, respectivamente.

5.3.5 Análise de escalabilidade do protótipo

Para testar a escalabilidade do protótipo, foram realizados experimentos em um ambiente com 32 máquinas. A *workload* definida tinha 8 gigabytes de tamanho, resultando em 4 *chunks* alocados para cada nó.

É importante salientar que nessa análise será apenas verificado o comportamento do protótipo sendo executado neste ambiente. O objetivo é avaliar como é tratado o aumento de ambiente e de *workload* pelo protótipo, realizando relações com as experimentações anteriores sempre que pertinente.

Esta experimentação teve apenas 6 replicações, pois o objetivo dela é só de certificar que a arquitetura é escalável. Adicionalmente, pode-se também analisar o comportamento do protótipo processando uma quantidade de dados mais próxima do que se é utilizado normalmente em *jobs* MapReduce.

Os fatores utilizados foram os mesmos das experimentações anteriores. Os níveis inicialmente foram definidos de acordo com a Tabela 5.4, no entanto, o protótipo apresentou problemas de execução quando foram configuradas 122 tarefas de *Reduce*. Estes problemas ocorreram durante a transferência de dados intermediários. Não foram investidos esforços para diagnosticar com exatidão quais foram as causas. Esse processo apresenta

diversos pontos pouco eficientes e para um novo *framework*, ele vai ser completamente reimplementado. Portanto, o segundo nível de tarefas de redução foi definido como 2 tarefas alocadas para cada *peer*, totalizando 64 (acima desse número, o tempo de execução começou a ficar proibitivo).

Na Figura 5.13 pode-se visualizar o tempo total do *job* desta configuração. Note que o comportamento apresentado é bem semelhante aos resultados apresentados na Seção 5.3.1, o que é justo pois em ambos são elencados 4 *chunks* por máquina.

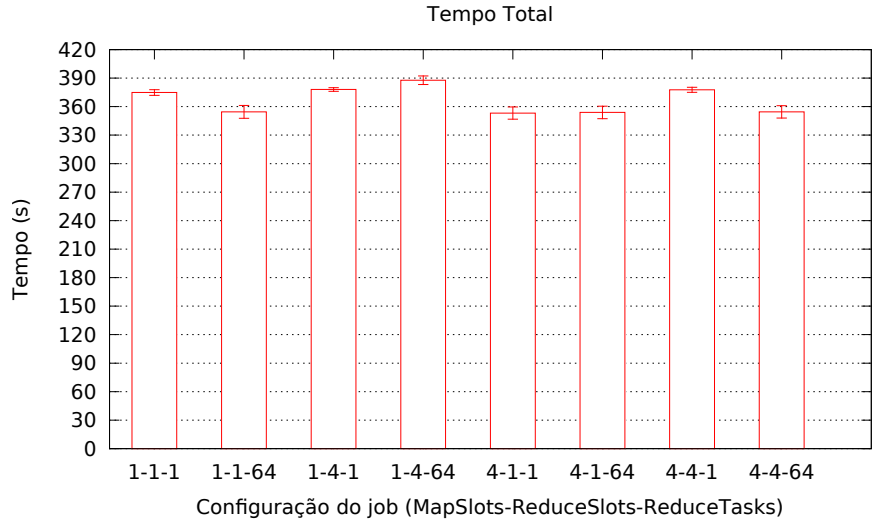

Figura 5.13: Tempo total do *job*, processando 8 gigabytes em 32 máquinas.

Os resultados utilizando todas as variáveis de resposta disponíveis no protótipo podem ser visualizados na Seção A.4. O *Map* (Figura A.19) mostrou tempos e resultados similares ao experimento realizado em 3 máquinas processando 768 megabytes. Isso se deve ao fato de que em ambos os casos cada *peer* tem a mesma quantidade de dados para processamento.

A fase de *Reduce* (Figura A.20) apresentou tempos bem maiores que o experimento referenciado. Como há uma quantidade muito maior de pares intermediários a serem transmitidos, o gargalo acaba impactando mais no *job*.

O processamento de *Reduces* (Figura A.21) apresentou uma queda muito grande, ao variar-se o número de reduções. Como havia 8 gigabytes de dados de entrada, quando utilizou-se apenas uma tarefa, o tempo foi muito maior comparado com o obtido utilizando-se 64. Isso é resultado da distribuição dos dados para processamento. No segundo caso, divide-se aquela grande redução em 64 tarefas menores, processadas em paralelo por todos os *peers* no ambiente, resultando na duração de execução observada.

Portanto, pode-se concluir que o modelo em si é escalável. Os problemas de execução citados são relacionados à implementação do protótipo. Não investigou-se detalhadamente suas causas pois na implementação do novo *framework* utilizando a arquitetura Maresia, toda a transmissão de dados será reimplementada, tendo em vista a quantidade de problemas apresentados pela técnica atualmente utilizada.

5.4 Considerações Finais

Neste Capítulo, foram realizadas as avaliações propostas neste trabalho. Tratando-se da análise comportamental, foram identificados os gargalos do protótipo. Nesta avaliação,

as principais conclusões obtidas foram as seguintes:

- No processamento da fase de *Map*, obtém-se resultados com um comportamento homogêneo. Isso mostra que este processo está muito bem implementado e que provavelmente não precisará de esforços adicionais de otimização na implementação do novo *framework*.

- O protótipo sofre uma degradação de desempenho muito grande (vide resultados) devido à transmissão de dados intermediários. Diversos pontos na implementação são citados como tendo influência sobre isso.

- O processamento de tarefas de *Reduce* não está implementado de forma eficiente. Dois pontos na implementação contribuem para isso (atraso na criação de *threads* e criação de *threads* sem tarefas associadas).

A avaliação comparativa do protótipo com o Hadoop tinha o objetivo de verificar se essa nova arquitetura poderia ser competitiva com o modelo mais tradicional (Cliente/Servidor). Desta experimentação, foram tiradas as seguintes conclusões:

- O mapeamento no protótipo é mais eficiente que no Hadoop, reforçando que provavelmente não será necessário esforços adicionais de otimização neste processo. Este resultado foi atribuído à linguagem de programação utilizada (protótipo utiliza C, Hadoop utiliza Java) e ao tratamento de saídas do map.

- O Hadoop, em geral, é mais eficiente que o protótipo. Isso é resultado de uma transferência de dados intermediários eficiente.

- Na tentativa de comprovar que é a transmissão de dados intermediários que fazem com que o Hadoop seja muito mais eficiente que o protótipo, foi realizado uma experimentação com uma *workload* sintética. Ela possuía uma pequena variabilidade de palavras, o que diminui a quantidade de pares a serem transmitidos. Neste experimento, foram obtidos casos onde o protótipo é mais eficiente, reforçando a conclusão do item anterior.

A última experimentação tinha o objetivo de avaliar a escalabilidade da arquitetura. Para isso, foram executados testes sobre um ambiente com uma grande quantidade de nós. Nesta análise, foi verificado o seguinte:

- Durante a primeira tentativa de experimentação, foram utilizadas 122 reduções. O tempo para transmissão de dados intermediários foi proibitivo, fazendo com que fosse abortada a avaliação com esse número. Ao reduzí-lo para 64 reduções (no ambiente, haviam 32 máquinas resultando em 2 tarefas por máquina), o *job* foi executado em um tempo aceitável.

- Ao executar-se experimentos com mais de 2 tarefas por máquina, o tempo de execução do *job* era proibitivo.

- Isso destaca mais ainda os problemas com a transmissão de dados. Não foram investigadas as causas do ocorrido, apenas foi definido que no novo *framework*, este processo será completamente reimplementado.

- Apesar disso, os processos locais de execução (mapeamento, processamento de reduções) tiveram um bom desempenho. Portanto, pode-se concluir que a arquitetura é escalável, no entanto, deve-se implementar a transferência de dados intermediários de forma eficiente.

Com isso, concluiu-se que a arquitetura possui potencial de ser tão eficiente quanto os modelos tradicionais. Além disso, foram destacados diversos pontos que devem ter atenção especial na nova implementação proposta.

6 CONCLUSÃO

Em (MARCOS, 2013) o protótipo da arquitetura Maresia foi desenvolvido e validado com o objetivo de mostrar a viabilidade de se executarem *jobs* MapReduce sobre ele. Como sua validação foi bem sucedida, será desenvolvido um novo *framework* que deve implementar tudo o que foi proposto na arquitetura, pois houveram algumas simplificações no protótipo.

Antes de iniciar essa implementação ainda era necessário avaliar alguns pontos no protótipo. Estes foram considerados como objetivos deste trabalho, são eles:

- Analisar seu comportamento, identificando pontos que necessitem de maior atenção nesta nova implementação;
- Realizar um comparativo com o Hadoop, para verificar se seu tempo de execução pode ser competitivo com as arquiteturas tradicionais;
- Avaliar a escalabilidade da arquitetura, realizando-se testes com um grande número de nós.

Além disso, foram realizadas modificações no protótipo da arquitetura Maresia que permitiram que ele explorasse o paralelismo local. Isso fez com que os fluxos de execução dos *frameworks* fossem mais semelhantes e possibilitou uma análise comparativa mais completa.

Sobre a análise comportamental do protótipo, sem dúvida foi alcançado o objetivo de identificar pontos que necessitem de maior atenção na nova implementação. O maior gargalo observado foi a transferência de dados intermediários e três características contribuem significativamente para a degradação de seu desempenho, são elas:

- Metodologia de transmissão de pares: Estes são enviados na forma de um par por mensagem. Cada lista resultante de uma tarefa de *Map*, na *workload* obtida dos dumps da Wikipedia gerava listas com dezenas de milhares de pares. Portanto, há dezenas de milhares de vezes o *overhead* de cabecalhos TCP/IP, além de ser um processo lento. Isso pode ser otimizado enviando-se mais de um par por mensagem (tendo em vista que pode-se enviar até 1460 bytes de conteúdo em uma mensagem TCP);
- Criação e destruição de conexões TCP a cada transferência de lista: Para cada lista saída do mapeamento, é criada uma conexão TCP para o envio de dados intermediários. Otimiza-se esse processo, mantendo as conexões criadas até o final da transmissão, evitando de ela precisar ser recriada, caso necessite enviar mais dados para aquele nó;

- Fluxo de envio/recebimento de dados: Tem grande contribuição na performance apresentada. Dentro deste fluxo, há uma região crítica, onde pode ocorrer a serialização do recebimento das mensagens. Além disso, ele inclui o envio de um ACK em nível de aplicação para aquele que enviou o par intermediário, dobrando o número de mensagens trafegadas. Enquanto este ACK não chega no remetente, ele não envia novos dados. A utilização da confiabilidade do protocolo de transporte e a eliminação desta região crítica otimiza este processo.

Além disso, foram constatados problemas de implementação também na execução de tarefas de redução. Foi verificado um atraso na criação de *threads*, além de o sistema sempre criar o número configurado de *slots* mesmo que eles não tenham tarefas para processar.

No comparativo entre os *frameworks*, o protótipo teve uma performance superior na fase de mapeamento. Isso é devido à linguagem de programação utilizada e ao tratamento dos dados intermediários no protótipo ser todo em memória, enquanto no Hadoop é utilizada gravação em disco. Considerando-se o *job* como um todo, o Hadoop foi mais eficiente pois o desempenho do protótipo é muito degradado em decorrência de seus gargalos.

Verificou-se também, com experimentos com uma *workload* gerada sinteticamente, que o protótipo em casos onde havia apenas uma tarefa de *Reduce* foi mais eficiente que o Hadoop na execução completa do *job*. Isso acontece pois o protótipo gerencia melhor a transferência de dados intermediários com apenas uma redução, quando foi definido o segundo nível deste fator (12 reduções), o Hadoop voltou a ser mais eficiente na execução total do *job*.

Na avaliação de escalabilidade da arquitetura, o protótipo foi executado em um ambiente com 32 nós e com 8 gigabytes de entrada. A transferência de dados intermediários apresentou alguns problemas, limitando o número de reduções que foi possível definir no sistema. Ao restringir o número de reduções para 64, o *job* foi executado com sucesso nesse ambiente e verificou-se que a arquitetura é escalável. No entanto, o *framework* que implementa a arquitetura deve ter uma transferência de pares intermediários otimizada para que essa execução seja eficiente.

Portanto, Maresia tem potencial para apresentar uma eficiência tão boa ou até melhor que os modelos *Master/Worker*. Além disso, ela trata problemas de pontos únicos de falha em nível de arquitetura, resultando em uma maior confiabilidade para o *framework* que a implementa. Para que o potencial seja atingido, no novo *framework* a ser desenvolvido deve-se investir na implementação da transmissão de dados intermediários de forma eficiente, pois ela impacta muito na execução de *jobs* MapReduce.

REFERÊNCIAS

ARANTES, L.; SOPENA, J. Garantindo a Circulacao e Unicidade do Token em Algoritmos com Nós Organizados em Anel Sujeitos a Falhas. In: WORKSHOP DE TOLERâNCIA A FALHAS (WTF), 2010, 2010. **Proceedings...** [S.l.: s.n.], 2010. p.17–30.

BAER, J. L. **Microprocessor Architecture**: from simple pipelines to chip multiprocessors. [S.l.]: Cambridge University Press, 2009. 9–12p.

CLEMENT, A.; KAPRITSOS, M.; LEE, S.; WANG, Y.; ALVISI, L.; DAHLIN, M.; RICHE, T. Upright cluster services. In: ACM SIGOPS 22ND SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES, 2009, New York, NY, USA. **Proceedings...** ACM, 2009. p.277–290. (SOSP '09).

CORPORATE, C. **Facebook irá usar ćold storageṕara armazenar a sua vasta quantidade de dados**. Disponível em: <http://corporate.canaltech.com.br/noticia/facebook/Facebook-ira-usar-cold-storage-para-armazenar-a-sua-vasta-quantidade-de-dados/>. Acesso em: Junho 2014.

DATASTAX. **Getting Started with Brisk**: hadoop powered by cassandra. Disponível em: <http://www.datastax.com/docs/0.8/brisk/index>. Acesso em: Julho 2013.

DEAN, J.; GHEMAWAT, S. MapReduce: simplified data processing on large clusters. In: SYMPOSIUM ON OPEARTING SYSTEMS DESIGN & IMPLEMENTATION - VOLUME 6, 6., 2004, Berkeley, CA, USA. **Proceedings...** USENIX Association, 2004. (OSDI'04).

DEAN, J.; GHEMAWAT, S. MapReduce: simplified data processing on large clusters. **Commun. ACM**, New York, NY, USA, v.51, n.1, p.107–113, 2008.

EMC. **EMC Home Page**. Disponível em: <http://www.emc.com/index.htm>. Acesso em: Maio 2013.

EMC. **EMC Digital Universe**. Disponível em: <http://www.emc.com/leadership/digital-universe/index.htm>. Acesso em: Maio 2013.

FANG, W.; HE, B.; LUO, Q.; GOVINDARAJU, N. Mars: accelerating mapreduce with graphics processors. **Parallel and Distributed Systems, IEEE Transactions on**, [S.l.], v.22, n.4, p.608–620, 2011.

GANTZ, J.; REINSEL, D. **THE DIGITAL UNIVERSE IN 2020**: big data ,bigger digital shadows, and biggest grow th in the far east. [S.l.]: IDC, 2012. White Paper.

GRID'5000. **Grid5000**: home - grid5000. Disponível em: <https://www.grid5000.fr/mediawiki/index.php/Grid5000:Home>. Acesso em: abril 2014.

GUNARATHNE, T.; WU, T.-L.; QIU, J.; FOX, G. MapReduce in the Clouds for Science. In: CLOUD COMPUTING TECHNOLOGY AND SCIENCE (CLOUDCOM), 2010 IEEE SECOND INTERNATIONAL CONFERENCE ON, 2010. **Anais...** [S.l.: s.n.], 2010. p.565–572.

HADOOP. **Welcome to Apache Hadoop!** Disponível em: <http://hadoop.apache.org/>. Acesso em: Maio 2013.

JAIN, R. K. **The Art of Computer Systems Performance Analysis**: techniques for experimental design, measurement, simulation, and modeling. [S.l.]: John Wiley & Sons, 1991. p.274–281.

LIU, H.; ORBAN, D. Cloud MapReduce: a mapreduce implementation on top of a cloud operating system. In: CLUSTER, CLOUD AND GRID COMPUTING (CCGRID), 2011 11TH IEEE/ACM INTERNATIONAL SYMPOSIUM ON, 2011. **Anais...** [S.l.: s.n.], 2011. p.464–474.

MARCOS, P. d. B. **Maresia - An Approach to Deal with the Single Points of Failure of the MapReduce Model**. 2013. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

MAROZZO, F.; TALIA, D.; TRUNFIO, P. A Framework for Managing MapReduce Applications in Dynamic Distributed Environments. In: PARALLEL, DISTRIBUTED AND NETWORK-BASED PROCESSING (PDP), 2011 19TH EUROMICRO INTERNATIONAL CONFERENCE ON, 2011. **Anais...** [S.l.: s.n.], 2011. p.149–158.

MCCRUM, R.; MACNEIL, R.; CARN, W. **The Story of English**: third revised edition. [S.l.]: Penguin Books; Revised edition (December 31, 2002), 2012. p.10.

MELLANOX. **Mellanox Infiniband Manual**. Disponível em: <http://www.mellanox.com/related-docs/prod_adapter_cards/PB_ConnectX_VPI.pdf>. Acesso em: Junho 2014.

PIERCE, J. R. **An Introduction to Information Theory**: symbols, signals and noise. [S.l.]: Dover Publications; Subsequent edition (March 29, 2012), 2012. p.75.

RANGER, C.; RAGHURAMAN, R.; PENMETSA, A.; BRADSKI, G.; KOZYRAKIS, C. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In: IEEE 13TH INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE COMPUTER ARCHITECTURE, 2007., 2007, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2007. p.13–24. (HPCA '07).

SANGROYA, A.; SERRANO, D.; BOUCHENAK, S. Benchmarking Dependability of MapReduce Systems. In: IEEE INT. SYM. ON RELIABLE DISTRIBUTED SYSTEMS (SRDS), 2012. **Anais...** [S.l.: s.n.], 2012.

SHVACHKO, K.; KUANG, H.; RADIA, S.; CHANSLER, R. The Hadoop Distributed File System. In: MASS STORAGE SYSTEMS AND TECHNOLOGIES (MSST), 2010 IEEE 26TH SYMPOSIUM ON, 2010. **Anais...** [S.l.: s.n.], 2010. p.1–10.

STOICA, I.; MORRIS, R.; KARGER, D.; KAASHOEK, M. F.; BALAKRISHNAN, H. Chord: a scalable peer-to-peer lookup service for internet applications. In: APPLICATIONS, TECHNOLOGIES, ARCHITECTURES, AND PROTOCOLS FOR COMPUTER COMMUNICATIONS, 2001., 2001, New York, NY, USA. **Proceedings...** ACM, 2001. p.149–160. (SIGCOMM '01).

STUDENT. Probable Error of a Correlation Coefficient. **Biometrika**, [S.l.], v.6, n.2–3, p.302–310, September 1908.

VENNER, J. **Pro Hadoop**. Berkely, CA, USA: Apress, 2009.

WANG, F.; QIU, J.; YANG, J.; DONG, B.; LI, X.; LI, Y. Hadoop high availability through metadata replication. In: CLOUD DATA MANAGEMENT, 2009, New York, NY, USA. **Proceedings...** ACM, 2009. p.37–44. (CloudDB '09).

WHITE, T. **Hadoop**: the definitive guide. [S.l.]: O'Reilly Media, Inc., 2009.

WIKI, H. **How Many Maps and Reduces**. Disponível em: <http://wiki.apache.org/hadoop/HowManyMapsAndReduces>. Acesso em: Maio 2014.

WIKIPEDIA Dumps. Disponível em: <http://meta.wikimedia.org/wiki/Data_dumps>. Acesso em: Abril 2014.

APÊNDICE A ANEXO DE FIGURAS

A.1 Resultados Configuração II

Os resultados aqui sumarizados, têm confiabilidade de 90% para 6 repetições e 95% de confiabilidade para 30 repetições.

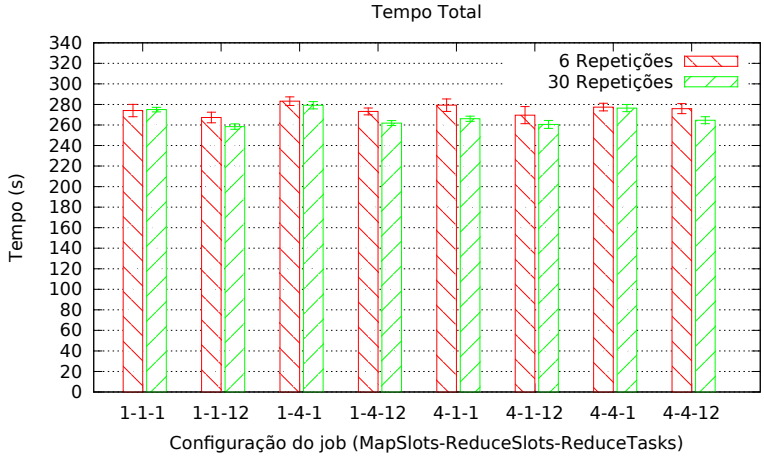

Figura A.1: Médias e intervalos de confiança do Tempo total do *job*, processando 1536 megabytes em 3 máquinas.

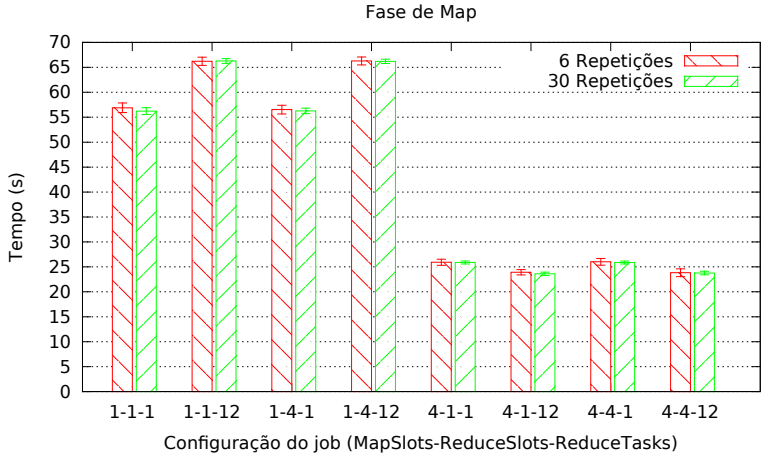

Figura A.2: Médias e intervalos de confiança da fase de *Map*, processando 1536 megabytes em 3 máquinas.

Figura A.3: Médias e intervalos de confiança da fase de *Reduce*, processando 1536 megabytes em 3 máquinas.

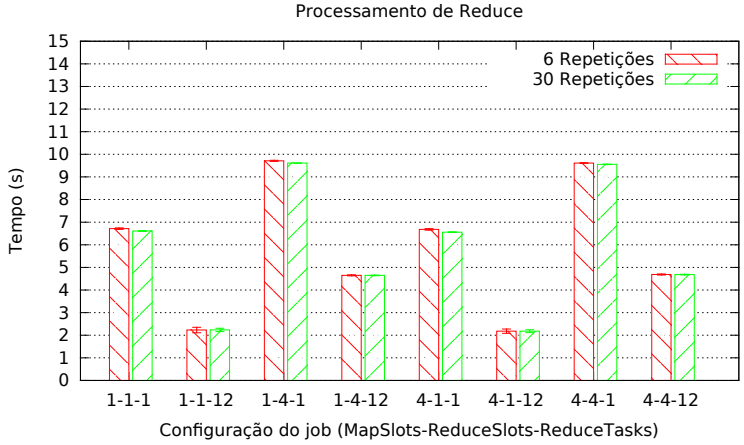

Figura A.4: Médias e intervalos de confiança do processamento da fase de *Reduce* (desconsiderando a transferência de dados intermediários), processando 1536 megabytes em 3 máquinas.

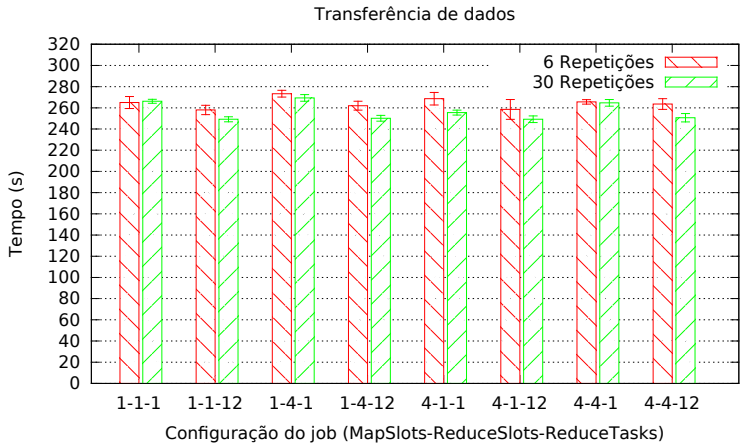

Figura A.5: Médias e intervalos de confiança da transferência de dados intermediários, processando 1536 megabytes em 3 máquinas.

A.2 Resultados Configuração III

Os resultados aqui sumarizados, têm confiabilidade de 90% para 6 repetições e 95% de confiabilidade para 30 repetições.

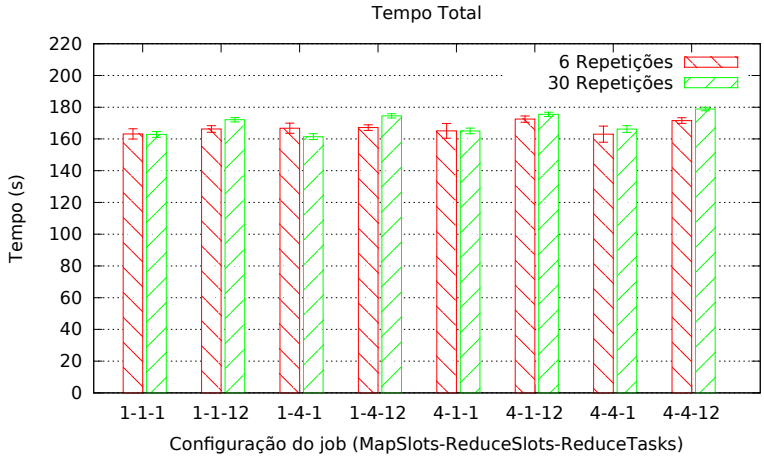

Figura A.6: Médias e intervalos de confiança do Tempo total do *job*, processando 1536 megabytes em 6 máquinas.

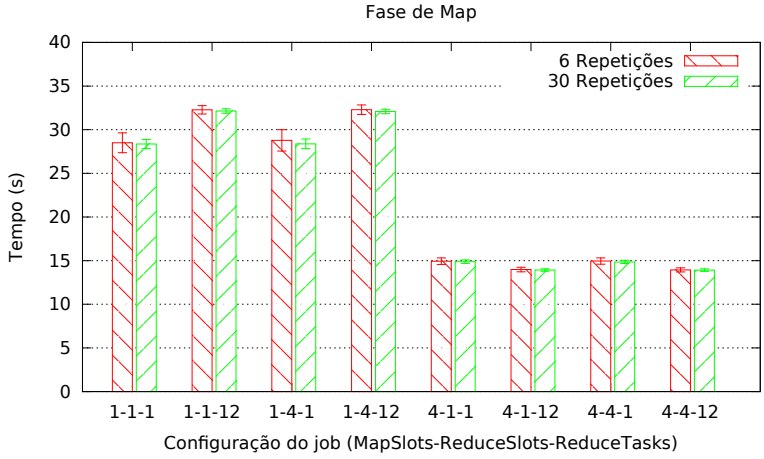

Figura A.7: Médias e intervalos de confiança da fase de *Map*, processando 1536 megabytes em 6 máquinas.

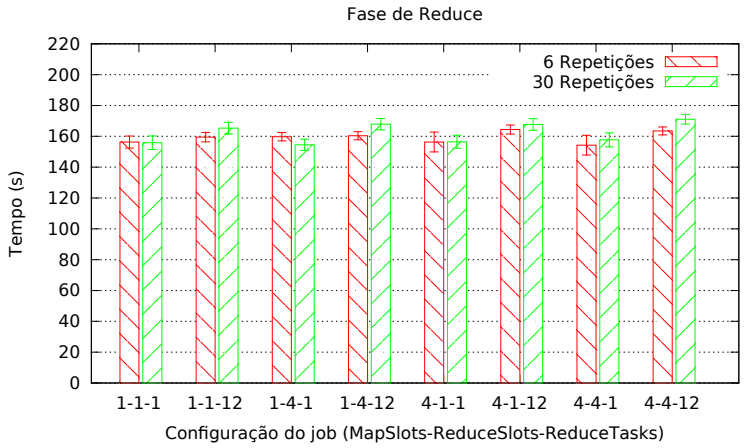

Figura A.8: Médias e intervalos de confiança da fase de *Reduce*, processando 1536 megabytes em 6 máquinas.

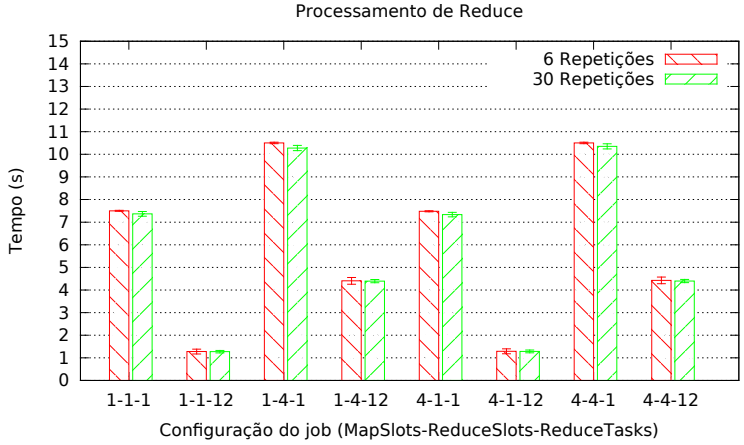

Figura A.9: Médias e intervalos de confiança do processamento da fase de *Reduce* (desconsiderando a transferência de dados intermediários), processando 1536 megabytes em 6 máquinas.

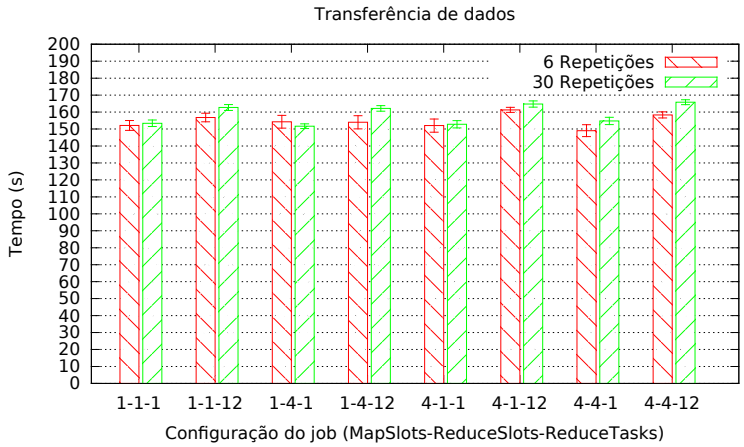

Figura A.10: Médias e intervalos de confiança da transferência de dados intermediários, processando 1536 megabytes em 6 máquinas.

A.3 Resultados análise comparativa

Os resultados aqui sumarizados, têm confiabilidade de 95%. Tanto para o Hadoop quanto para o protótipo, foram utilizadas 30 repetições nos testes.

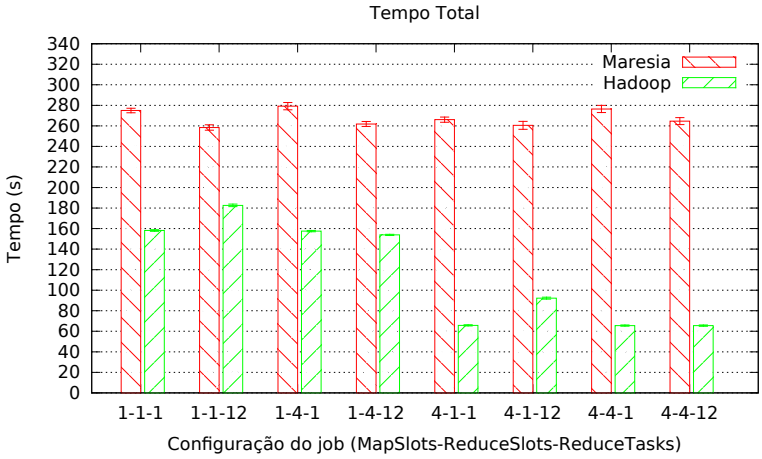

Figura A.11: Comparação entre tempos de execução do protótipo e do Hadoop, processando 1536 megabytes em 3 máquinas.

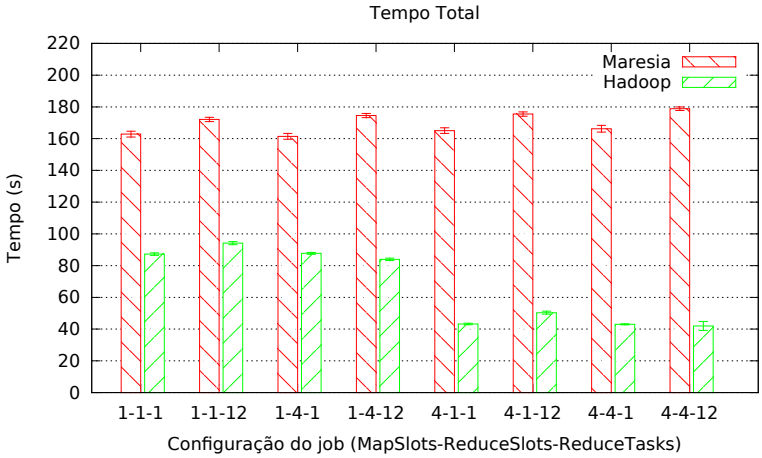

Figura A.12: Comparação entre tempos de execução do protótipo e do Hadoop, processando 768 megabytes em 6 máquinas.

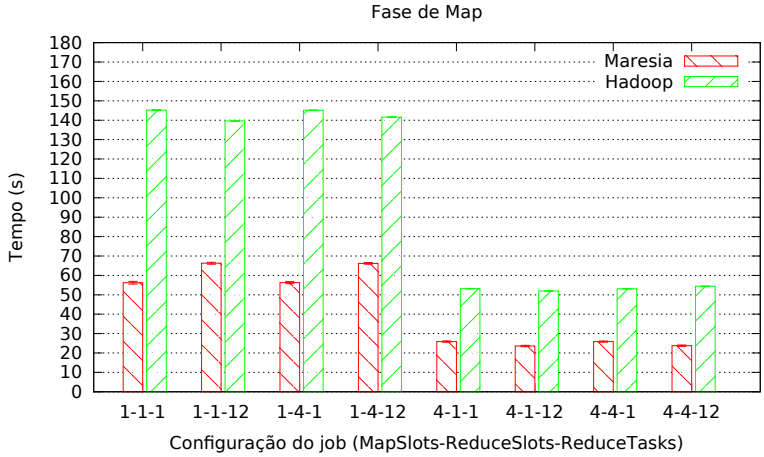

Figura A.13: Comparação entre tempos de execução da fase de *Map* do protótipo e do Hadoop, processando 1536 megabytes em 3 máquinas.

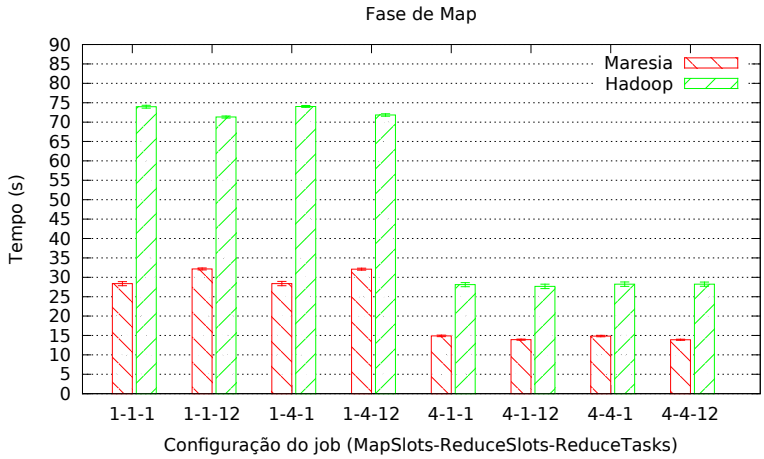

Figura A.14: Comparação entre tempos de execução da fase de *Map* do protótipo e do Hadoop, processando 1536 megabytes em 6 máquinas.

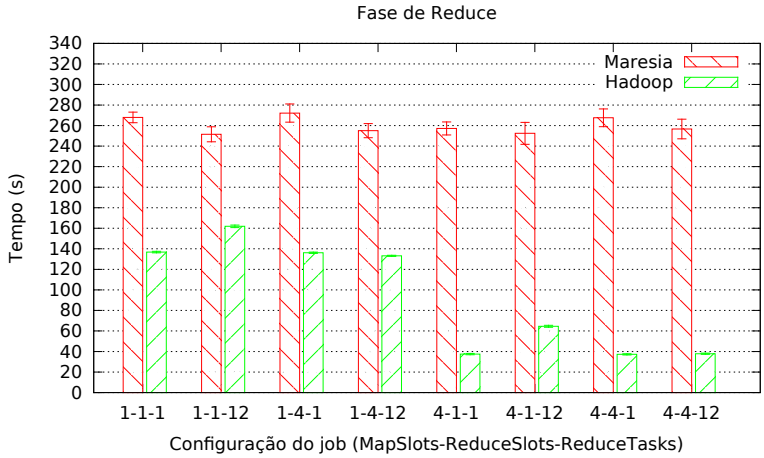

Figura A.15: Comparação entre tempos de execução da fase de *Reduce* do protótipo e do Hadoop, processando 1536 megabytes em 3 máquinas.

Figura A.16: Comparação entre tempos de execução da fase de *Reduce* do protótipo e do Hadoop, processando 1536 megabytes em 6 máquinas.

Figura A.17: Comparação da duração da fase de *Map* entre Hadoop e protótipo, processando uma *workload* sintética de 768 megabytes em 3 máquinas.

Figura A.18: Comparação da duração da fase de *Reduce* entre Hadoop e protótipo, processando uma *workload* sintética de 768 megabytes em 3 máquinas.

A.4 Resultados da análise de escalabilidade

Os resultados aqui sumarizados, têm confiabilidade de 90%. Foram utilizadas 6 repetições nos testes.

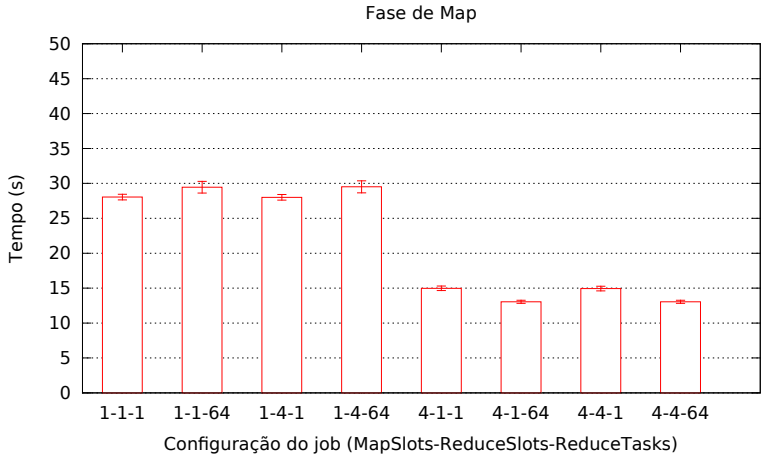

Figura A.19: Médias e intervalos de confiança da fase de *Map*, processando 8 gigabytes em 32 máquinas.

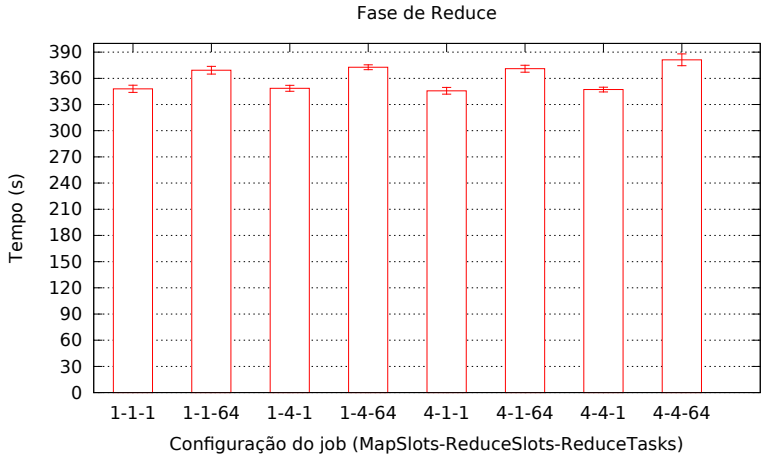

Figura A.20: Médias e intervalos de confiança da fase de *Reduce*, processando 8 gigabytes em 32 máquinas.

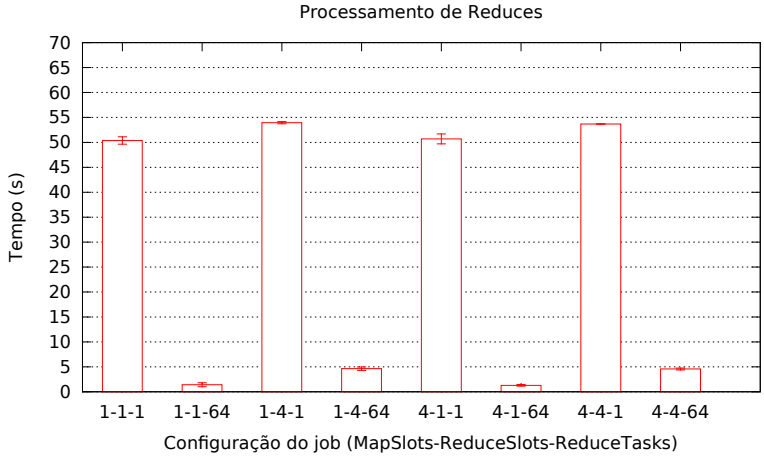

Figura A.21: Médias e intervalos de confiança da fase de *Reduce* (excluindo-se a transferência de dados), processando 8 gigabytes em 32 máquinas.

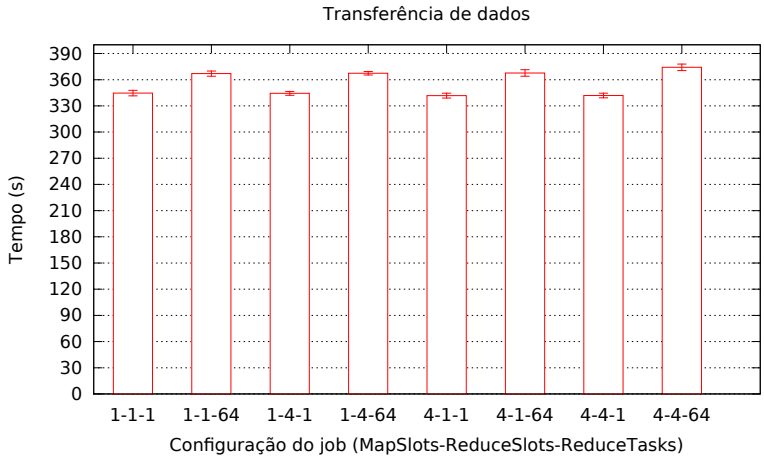

Figura A.22: Médias e intervalos de confiança da transferência de pares intermediários, processando 8 gigabytes em 32 máquinas.

APÊNDICE B ANEXO DE TABELAS

B.1 Alocações de variação - Configuração II

Os identificadores dos fatores sumarizados nos resultados aqui exibidos, são definidos na Tabela 5.2.

Fator	Alocação de Variação 6 Repetições	Alocação de Variação 30 Repetições
A	0,3926130944%	0,5962945694%
B	7,3740740578%	6,355189601%
C	15,136688118%	34,1204256337%
AB	2,163760839%	0,6166047029%
AC	0,5932597815%	3,5348242975%
BC	0,4751174424%	0,6560522005%
ABC	2,5378767959%	0,3834699671%
Erros	71,326609871%	53,7371390278%

Tabela B.1: Alocações de variação geradas com um experimento 2^kr, para o tempo total do *job*.

Fator	Alocação de Variação 6 Repetições	Alocação de Variação 30 Repetições
A	96,4951324486%	96,1225378522%
B	0,000483281%	4,331763655782E-005%
C	0,9990505869%	1,1033794134%
AB	0,0004624174%	0,0002124152%
AC	2,447658665%	2,6861766415%
BC	0,0002699324%	9,23120828937653E-006%
ABC	0,0016104666%	0,0004161663%
Erros	0,0553322021%	0,0872249626%

Tabela B.2: Alocações de variação geradas com um experimento 2^kr, para a fase de *Map*.

Fator	Alocação de Variação 6 Repetições	Alocação de Variação 30 Repetições
A	0,0001661938%	2,1106773057%
B	3,6336650674%	6,4495573474%
C	9,1230305183%	31,38937119%
AB	1,7993158483%	0,6161597958%
AC	0,2482339974%	4,131754777%
BC	0,5625596988%	0,6086063576%
ABC	2,1015974924%	0,387943384%
Erros	82,5314311836%	54,3059298426%

Tabela B.3: Alocações de variação geradas com um experimento 2^kr, para a fase de *Reduce*.

Fator	Alocação de Variação 6 Repetições	Alocação de Variação 30 Repetições
A	0,0079289243%	0,0060545885%
B	24,4671919845%	25,1921515241%
C	75,0491465384%	73,815623639%
AB	0,0006900188%	0,004396242%
AC	0,0008488864%	0,0003897018%
BC	0,3264939803%	0,4291976449%
ABC	0,0075081614%	0,0053658164%
Erros	0,140191506%	0,5468208433%

Tabela B.4: Alocações de variação geradas com um experimento 2^kr, utilizando a duração do processamento da fase de *Reduce* como variável de resposta (desconsiderando a transferência de dados intermediários).

Fator	Alocação de Variação 6 Repetições	Alocação de Variação 30 Repetições
A	0,1101689511%	2,2391449115%
B	3,303528917%	3,3948910912%
C	15,2518324324%	34,0220739431%
AB	2,2279307174%	0,5678781565%
AC	0,6686212788%	3,8806642421%
BC	0,0978246661%	0,9578453187%
ABC	2,8284111155%	0,3883451556%
Erros	75,5116819218%	54,5491571814%

Tabela B.5: Alocações de variação geradas com um experimento 2^kr, utilizando a transferência de dados intermediários como variável de resposta.

B.2 Alocações de variação - Configuração III

Os identificadores dos fatores sumarizados nos resultados aqui exibidos, são definidos na Tabela 5.2.

Fator	Alocação de Variação 6 Repetições	Alocação de Variação 30 Repetições
A	4,3439185167%	6,4757342215%
B	0,118655995%	0,9341147045%
C	21,2591508578%	61,7441137915%
AB	3,1253501902%	0,3638104723%
AC	8,6825301382%	0,0142691553%
BC	0,132322171%	1,0864133626%
ABC	0,8326496946%	0,0853840778%
Erros	61,5054224365%	29,2961602146%

Tabela B.6: Alocações de variação geradas com um experimento 2^kr, para o tempo total do *job*.

Fator	Alocação de Variação 6 Repetições	Alocação de Variação 30 Repetições
A	97,2095873405%	97,0654622604%
B	0,0016259948%	0,0001625628%
C	0,6707734258%	0,7537310709%
AB	0,0023400944%	3,82162845990619E-006%
AC	2,0342361946%	2,1311681286%
BC	0,0022431686%	5,45003396315321E-005%
ABC	0,0009697094%	0,0003291106%
Erros	0,0782240719%	0,0490885447%

Tabela B.7: Alocações de variação geradas com um experimento 2^kr, para a fase de *Map*.

Fator	Alocação de Variação 6 Repetições	Alocação de Variação 30 Repetições
A	0,3639971093%	2,5486275143%
B	0,1165831469%	1,0100723513%
C	24,9317733302%	66,0174144301%
AB	2,9191010886%	0,3663769059%
AC	10,2099801857%	0,0899763601%
BC	0,1134448665%	1,113930944%
ABC	0,7895364098%	0,0948981543%
Erros	60,5555838631%	28,7587033399ss%

Tabela B.8: Alocações de variação geradas com um experimento 2^kr, para a fase de *Reduce*.

Fator	Alocação de Variação 6 Repetições	Alocação de Variação 30 Repetições
A	0,0079289243%	0,000808035%
B	24,4671919845%	21,5919095163%
C	75,0491465384%	77,9432207653%
AB	0,0006900188%	0,0006606164%
AC	0,0008488864%	5,03425318050761E-006%
BC	0,3264939803%	0,0073488804%
ABC	0,0075081614%	0,0031706477%
Erros	0,140191506%	0,4528765046%

Tabela B.9: Alocações de variação geradas com um experimento 2^kr, utilizando a duração do processamento da fase de *Reduce* como variável de resposta (desconsiderando a transferência de dados intermediários).

Fator	Alocação de Variação 6 Repetições	Alocação de Variação 30 Repetições
A	0,1227312091%	2,6207577858%
B	2,0617747475%	0,0721448807%
C	14,5570982323%	62,3599789019%
AB	2,9768612199%	0,3435743368%
AC	9,0031428591%	0,0520492688%
BC	2,6261538913%	0,0424115971%
ABC	1,0146113324%	0,1009742903%
Erros	67,6376265083%	34,4081089386%

Tabela B.10: Alocações de variação geradas com um experimento 2^kr, utilizando a transferência de dados intermediários como variável de resposta..

B.3 Alocações de variação - Análise comparativa

Os identificadores dos fatores sumarizados nos resultados aqui exibidos, são definidos na Tabela 5.2.

Fator	Maresia	Hadoop
A	0,5962945694%	93,7767283099%
B	6,355189601%	2,2570880702%
C	34,1204256337%	1,5887833777%
AB	0,6166047029%	0,0038802168%
AC	3,5348242975%	0,0230295344%
BC	0,6560522005%	2,1274359925%
ABC	0,3834699671%	0,001674331%
Erros	53,7371390278%	0,2213801675%

Tabela B.11: Comparação de alocações de variação entre Hadoop e o protótipo, para o tempo total do *job*, processando 1536 megabytes em 3 máquinas.

Fator	Maresia	Hadoop
A	6,4757342215%	95,2420738409%
B	0,9341147045%	1,0428492308%
C	61,7441137915%	0,2645058883%
AB	0,3638104723%	0,0044131587%
AC	0,0142691553%	0,0278704721%
BC	1,0864133626%	1,114826032%
ABC	0,0853840778%	0,021295687%
Erros	29,2961602146%	2,2821656902%

Tabela B.12: Comparação de alocações de variação entre Hadoop e o protótipo, para o tempo total do *job*, processando 1536 megabytes em 6 máquinas.

Fator	Maresia	Hadoop
A	96,1225378522%	99,8297197672%
B	4,331763655782E-005%	0,0144774634%
C	1,1033794134%	0,0636270848%
AB	0,0002124152%	0,0001857715%
AC	2,6861766415%	0,0667696783%
BC	9,23120828937653E-006%	0,0154813048%
ABC	0,0004161663%	0,0001381878%
Erros	0,0872249626%	0,0096007421%

Tabela B.13: Comparação de alocações de variação entre Hadoop e o protótipo, para a fase de *Map* processando 1536 megabytes em 3 máquinas.

Fator	Maresia	Hadoop
A	97,0654622604%	99,5383942404%
B	0,0001625628%	0,005056179%
C	0,7537310709%	0,0872592631%
AB	3,82162845990619E-006%	4,52183841103767E-005%
AC	2,1311681286%	0,0607734433%
BC	5,45003396315321E-005%	0,0025625041%
ABC	0,0003291106%	0,00000052%
Erros	0,0490885447%	0,3059086318%

Tabela B.14: Comparação de alocações de variação entre Hadoop e o protótipo, para a fase de *Map* processando 1536 megabytes em 6 máquinas.