Uma Perspectiva da Computação Paralela no Modelo *MapReduce*

Marcelo Iury S. Oliveira{miso@cin.ufpe.br}, Felipe Alencar Lopes {fal3@cin.ufpe.br}

Centro de Informática da Universidade Federal de Pernambuco (CIn/UFPE)
Artigo escrito para a disciplina de Sistemas Distribuídos com foco em Programação Paralela (2014.1)
Facilitador: Fernando Castor {fjclf@cin.ufpe.br}

Resumo.

Novas formas de geração e aquisição de dados foram desenvolvidas, permitindo a expansão da chamada era da informação, marcada pelo aumento significativo do volume de dados a ser processado. Apesar desta demanda, grande parte dos computadores atuais não utiliza totalmente a sua capacidade de processamento. Ainda que suportem paralelismo e outras técnicas de execução, desenvolver sistemas que aproveitem todo o potencial destes computadores não é trivial. Diante da necessidade de processor quantidades de dados cada vez maiores, e da complexidade presente no desenvolvimento de aplicações para este fim, surgiu o modelo *MapReduce*. Este trabalho apresenta uma análise de pesquisas relativas ao *MapReduce*, focando esta análise em arquiteturas e implementações voltadas para a computação paralela.

1 Introdução

O desenvolvimento de novas formas de geração, aquisição de dados permitiu expansão da chamada era da informação, marcada pelo aumento significativo do volume de dados a ser processado e, assim como, da diminuição, também significativa, do tempo de transmissão da informação [1]. Muitos fatores contribuem para o aumento do volume de dados, tais como o crescimento de transações comerciais, o compartilhamento de conteúdo em mídias sociais, a coleta de dados por sensores, etc. Apesar de a capacidade de armazenamento não ser mais um fator limitador como era no passado, ainda permanecem os problemas da latência de transmissão das informações pelas redes de interligação e de processamento da informação.

Visto que a relação custo/desempenho de computadores caiu e que os atuais microprocessadores dispõem de múltiplas unidades de processamento em um único chip, atualmente não é difícil conectar computadores para viabilizar a construção de plataformas de processamento de dados. A ideia de se utilizar paralelismo é tão antiga quanto os computadores eletrônicos, tendo sido propostas uma grande variedade de soluções que fazem uso dos mais diversos tipos de arquiteturas, tais como, o uso oportunístico de máquinas [3], GPU [4], clusters de videogames [5] e computação na nuvem [6].

Diferentemente do modelo sequencial, a computação paralela não possui um modelo algorítmico que seja amplamente aceito. Um sistema de computação paralela consiste de uma coleção de processadores, não necessariamente do mesmo tipo, interconectados de acordo com uma determinada topologia para permitir a coordenação de suas atividades e troca de dados e, naturalmente, o desenvolvimento de novos algoritmos e programas, também paralelos. O desenvolvimento de aplicações paralelas não é uma tarefa trivial, pois frequentemente fica a cargo do programador lidar com questões relativa a controle de concorrência, distribuição e coordenação de atividades.

Diante da necessidade de processar grandes quantidade de dados e da complexidade de desenvolvimentos de aplicações voltadas para este fim, a Google propôs o modelo de programação paralela *MapReduce* [7]. O modelo de programação se baseia em duas funções fornecidas pelo usuário, uma de mapeamento dos dados (*i.e. map*) e redução dos dados (*i.e. reduce*). O sistema *MapReduce* gerencia o processamento através de um processo *master*, cuja função é de orquestrar o processamento e gerenciar o processo de agrupamento de dados e distribuir o processamento de forma equilibrada. A ideia do *MapReduce* é abstrair as dificuldades do ambiente paralelo para o programador, fornecendo uma interface simples de programação e oferecendo premissas de tolerância a falhas, distribuição de carga e comunicação dos dados.

O *MapReduce* é um modelo consolidado, tendo recebido atenção tanto da indústria quanto da academia. A versão original foi implementada utilizando C++, com interfaces para Java e Python [8]. Entretanto, outras implementações foram propostas desde primeira publicação do *MapReduce*. Há implementações disponíveis para Java (*e.g.* Hadoop [9]), .NET (*e.g.* Qizmt [10], MapSharp [11]), Ruby (*e.g.* Elastic [12]) e Python (*e.g.* Mrjob [13], Octopy [14]). Dentre estas plataformas, destaca-se o framework Hadoop que está sendo usado por grandes companhias, a exemplo de Adobe, Amazon, Facebook e Yahoo [15]. Embora o Hadoop seja desenvolvido em Java, seus programas não precisam ser codificados com Java, podendo ser usadas outras linguagens tais como Python, C++ e C# [9].

As implementações e evoluções do *MapReduce* não ficaram restritas ao suporte de diferentes linguagens de programação. Foram desenvolvidos novos algoritmos baseados no modelo *MapReduce* para dar suporte a vários de aplicações a exemplo de junção de dados [16] e algoritmos genéticos [17]. Também foram propostos trabalhos com o objetivo de melhorar o desempenho em fatores, tais como escalonamento de *jobs* [18] e consumo de energia [19]. No mais, a paralelização de tarefas não está restrita apenas ao uso de *clusters* de computadores. Há também uma quantidade significativa de pesquisas com o objetivo de adaptar o *MapReduce* para outras arquiteturas de computação paralela, por exemplo, unidades de processamento gráficas (do inglês, *Graphics Processing Unit* ou GPU), processadores *multicores* e *clusters* de computadores móveis.

Diante do exposto, o presente trabalho tem o objetivo de apresentar um *survey* de pesquisas relativas ao *MapReduce* com foco estudos voltados ao suporte ou migração do modelo de programação para novas arquiteturas de computação paralela. Este artigo está organizado da seguinte forma: a segunda seção apresenta uma visão geral do modelo *MapReduce*, a terceira seção elenca arquiteturas e implementações *MapReduce*, trazendo também o escopo da computação paralela. Exemplos de aplicações

são exibidos na quarta seção. As seções cinco e seis discutem o conteúdo apresentado e concluem este trabalho.

2 Visão Geral de *MapReduce*

Nos últimos anos, diversos modelos de programação surgiram para facilitar e permitir a utilização das arquiteturas com múltiplos processadores [20]. O *MapReduce* é um destes modelos, sendo utilizado para o processamento e a geração de grandes conjuntos de dados a partir de um algoritmo paralelo, distribuído em um *cluster* [21]. Programas escritos neste estilo de programação são automaticamente paralelizáveis. Esta seção irá definir e detalhar as principais características que compõem o modelo *MapReduce*, além de listar funcionalidades em comum dentre as implementações existentes deste modelo.

2.1 Modelo de Programação

Basicamente, o modelo *MapReduce* é composto por uma função *map*, que aplica filtros e realiza a ordenação nos dados (*e.g.* ordenar funcionários em filas utilizando o pelo primeiro nome, uma fila para cada nome), e uma função *reduce*, que realiza uma operação de síntese (*e.g.* contar o número de funcionários em cada fila, produzindo a frequência que cada nome aparece).

Este modelo é inspirado pelas funções de mesmo nome (*map* e *reduce*) comumente utilizadas no paradigma de programação funcional [21], embora o propósito destas funções implementadas no modelo *MapReduce* não seja o mesmo que suas formas originais [22]. É importante ressaltar que as contribuições chave da utilização do modelo *MapReduce* não são as funções *map* e *reduce*, mas o controle automático de escalabilidade, transferência de dados e tolerância a falhas, dentre outras funcionalidades. Estas contribuições são atingidas por uma variedade de aplicações através da otimização dos seus processos de execução. Entretanto, aplicar o modelo *MapReduce* em um sistema monoprocessado e monothread não oferece nenhum ganho de performance. Muito pelo contrário, a sobrecarga imposta pelo *MapReduce* torna impraticável usá-lo em ambientes monoprocessados. Somente quando uma operação qualquer distribuída é otimizada, diminuindo custos de comunicação na rede, e as funcionalidades de tolerância a falhas do modelo *MapReduce* vêm à tona, é que o uso deste modelo é benéfico.

Como já mencionado, têm sido implementadas diversas bibliotecas baseadas no *MapReduce*. Uma destas implementações é o Hadoop *MapReduce*, disponível livremente sob licença Apache [23]. O termo *MapReduce*, que originalmente se referia à tecnologia proprietária do Google, agora tem a implementação e o nome Hadoop, utilizado de forma genérica. Vale a pena mencionar que, apesar de existirem mais de uma implementação *MapReduce*, as bibliotecas geralmente têm as funções básicas do modelo de programação *MapReduce* (*map* e *reduce*) se comportando conforme mostra a Figura 1.

- *Map*: o nó *master* recebe o conjunto de entrada, realiza a divisão deste conjunto em subconjuntos menores, e os distribui para nós *worker*. Um *worker* pode realizar esta divisão novamente, levando a uma estrutura de árvore multi-nível. O *worker* processa o problema menor e envia o resultado para o nó *master* superior.
- *Shuffle*: é a denominação para as ações intermediárias entre o mapeamento e a redução dos dados.
- *Reduce*: O nó *master*, após a execução do mapeamento, então coleta as respostas para todos os subproblemas e os combina de tal forma que resulte numa saída composta por um novo conjunto de pares composto por chave/valor.

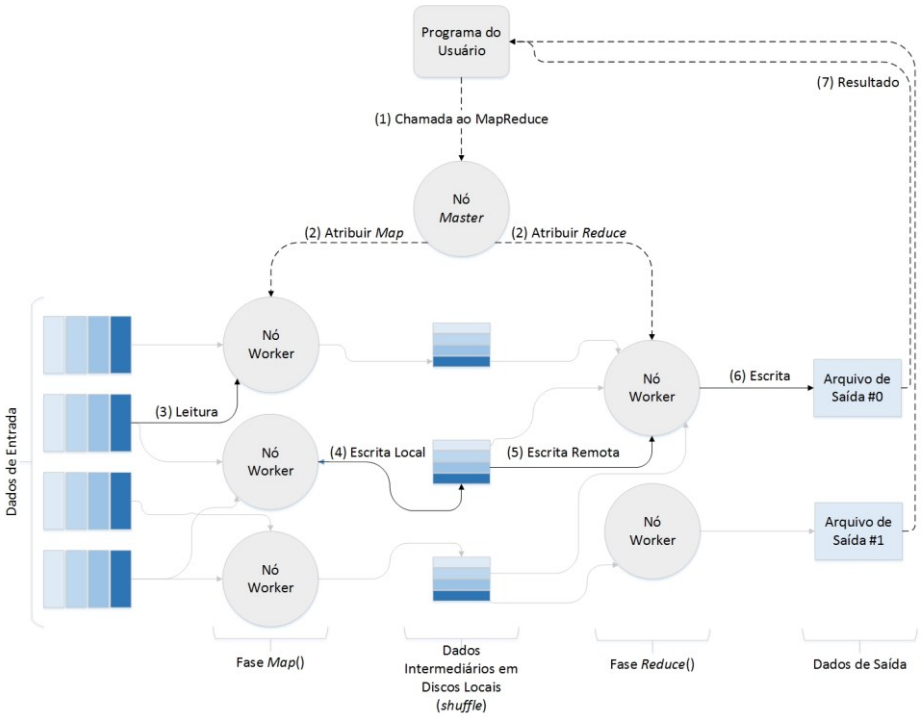

Fig. 1. Modelo de execução do *MapReduce*.

O *MapReduce* permite o processamento distribuído das operações *map* e *reduce*, desde que cada operação de mapeamento seja independente das outras. O trecho de código abaixo ilustra a implementação destas operações, o exemplo mostrado é encontrado na publicação original do *MapReduce*. O limite para o processamento distribuído, na prática, é o número de fontes de dados independentes e também a quantidade de CPUs disponíveis para realizar a operação. O mesmo acontece para o conjunto de operações *reduce*, que pode ser distribuída desde que todos os resultados do mapeamento que partilham a mesma chave sejam apresentados para o mesmo redutor ao mesmo tempo, ou que a função de redução seja associativa [7].

```
/* chave: nome do documento
* valor: elementos do documento*/
map(String key, String value){
  for each word w in value:
    emitIntermediate(w, "1");
}
/* key: uma palavra
* values: lista para contadores das palvaras*/
reduce (String key, Iterator values){
  int result = 0;
  for each v in values:
    result += ParseInt(v);
    emit(result);
}
```

2.2 Características

Desde a divulgação da primeira versão de uma implementação *MapReduce*, em 2003, diversas funcionalidades gerais foram identificadas para deste modelo [7], incluindo tolerância a falhas, balanceamento de carga, escalabilidade, performance, e a própria ordenação dos dados. Nesta sub-seção, são apresentadas as funcionalidades habilitadas pelo uso de uma infraestrutura *MapReduce*, expondo características das implementações *MapReduce* discutidas ao logo deste artigo.

Tolerância a falhas. Devido ao uso de *MapReduce* em grandes volumes dados, utilizando diversas máquinas para processar estes dados, a infraestrutura *MapReduce* deve tolerar falhas que possam ocorrer nestas máquinas, ou em algum momento do processamento, de forma satisfatória [21]. Geralmente, as infraestruturas *MapReduce* utilizam, essencialmente, dois mecanismos para tolerar falhas: (1) elas monitoram a execução das tarefas *map* e *reduce*, reinicializando-as caso elas parem de executar, e (2) estas infraestruturas adicionam *checksums* em conjuntos e subconjuntos de dados a fim de detectar falhas de integridade de dados [23] [24]. O tratamento automático de falhas é um benefício chave do *MapReduce*, escondendo do programador a complexidade da tolerância a falhas. Se um nó quebra, o modelo executa novamente as tarefas que estavam alocadas para este nó em outra máquina [25].

Balanceamento de carga. A funcionalidade de balanceamento de carga é utilizada para dar garantias de que nenhum dos recursos computacionais existentes estejam ociosos enquanto outros estiverem sendo utilizados. Para balancear a carga, o modelo *MapReduce* permite que a carga seja migrada entre *workers*, diminuindo o excesso de trabalho em alguns destes nó [26]. Quando este balanceamento é realizando durante a execução do sistema, ele é chamado *dynamic load balance* (balanceamento de carga dinâmico) - isto pode ser realizado tanto de maneira direta quanto de maneira iterativa de acordo com a seleção de execução do nó:

- Em métodos iterativos, o nó destino final é determinado através de diversos passos de iterações;
- Em métodos diretos, o nó destino final é selecionado em um passo.

Apesar do balanceamento de carga não ser tão significante na execução de um algoritmo *MapReduce*, ele se torna essencial quando se manipula arquivos grandes para processamento e quando o uso dos recursos de *hardware* está crítico. O balanceamento permite uma melhor utilização de recursos em situações críticas com um pequeno melhoramento na performance. Cada implementação *MapReduce* possui seu próprio algoritmo para balanceamento de carga. Devido a grande variedade de implementações, estes algoritmos não serão discutidos no presente artigo.

Escalabilidade. Para atingir a escalabilidade, um sistema computacional precisa permitir a execução de diversas *threads* simultâneas, isoladas umas das outras, ou, pelo menos, com mínima dependência entre execuções. O modelo *MapReduce* e a plataforma de execução distribuída foi originalmente descrita para facilmente escalar sistemas para executarem eficientemente através de milhares de máquinas em uma rede dedicada de alta velocidade. Um importante desenvolvimento do artigo original o Google é na aplicação do conceito *MapReduce* para o processamento paralelo em ambientes multicores, a escalabilidade atingida utilizando *MapReduce* para processar dados em grandes volumes de CPUs com baixos custos de implementação, seja em um único servidor ou múltiplas máquinas, é uma funcionalidade atrativa [22].

Apesar de sua arquitetura voltada para escalabilidade, as implementações *MapReduce* precisam lidar com dados que não podem ser plenamente paralelizados. Conforme definido anteriormente, para atingir a escalabilidade, um sistema de computador precisa permitir a execução de várias *threads* simultâneas (isoladas ou com mínima dependência entre execuções). Para possuir independência na execução, um sistema também precisa possuir independência no fornecimento de dados de entrada e saída. Se um destes tipos de dados precisa de sincronização entre as execuções simultâneas, então os benefícios da escalabilidade são severamente diminuídos. Este impacto é conhecido no paradigma de programação paralela como contenção.

Uma solução para superar esta contenção em dados compartilhados nas implementações *MapReduce* é empregar uma técnica conhecida como Particionamento de Dados. Esta técnica envolve o isolamento dos dados que são processados por cada execução em grupos, partições, as quais podem ser lidas e processadas independentemente de outras partições de dados. A técnica Particionamento de Dados é um princípio fundamental para permitir a escalabilidade [27].

Performance. A performance do modelo *MapReduce* foi avaliada, originalmente, para execuções *end-to-end* em sistemas [21]. Claramente, os ganhos obtidos com a utilização do *MapReduce* são provenientes do processamento paralelo e do paradigma de programação paralela. O balanceamento de carga, a técnica de particionamento de dados e a tolerância a falhas também compõem a característica performática do *MapReduce*, embora este não seja um ponto forte do modelo.

Outras características podem ser observadas nas variadas implementações *MapReduce* (*e.g.* armazenamento independente), mas nos limitamos a descrever aquelas comuns ao modelo original *MapReduce* [22].

2.3 Prós e Contras

Sistemas Gerenciadores de Bancos de Dados (SGBDs) adotaram uma estratégia para adaptá-los a diversos cenários e não são adequados para a resolução de tarefas de larga escala para o processamento de dados. Existe uma demanda para ferramentas de processamento de dados que sejam voltadas para tais problemas [28] [29] [30]. Enquanto o modelo *MapReduce* é referenciado como um novo caminho de processamento de dados para *big data* na computação de *data center*, ele também é criticado como retrocesso no processamento paralelo dos dados em comparação com os SGBDs [31] [32]. Entretanto, vários proponentes do *MapReduce* na indústria defendem que *MapReduce* não é um SGBD e que a comparação entre as duas soluções é injusta. Com o andamento da discussão técnica, em 2010 a Association for Computing Machinery (ACM) convocou ambos os lados para discutir essa questão na conferência Communications of the ACM (CACM) daquele ano [33] [34].

Dentre as comparações observadas, temos que: Pavlo *et al.* observou que a Hadoop (implementação *MapReduce*) é 2~50 vezes mais lenta que um SGBD paralelo, exceto no caso de carregamento de dados [35]. Anderson *et al.* também critica a baixa eficiência por nó obtida com uso da Hadoop para processamento paralelo de dados, apesar de demonstrar o ganho com escalabilidade desta implementação [36].

Em geral, os estudos demonstram uma clara troca entre eficiência e tolerância à falhas na utilização de implementações *MapReduce* em relação à SGBDs no processamento paralelo de dados. O *MapReduce* aumenta a tolerância à falhas em análises de longo prazo, através de *checkpoints* das tarefas completadas e replicação dos dados. Entretanto, as operações frequentes de entrada e saída para tolerância a falhas reduzem a eficiência do modelo.

SGBDs paralelos objetivam a eficiência no lugar da tolerância à falhas, realizando explorações ativas, canalizando resultados intermediários entre operações de consulta, busca, ou ordenação. Por outro lado, este foco na eficiência torna os SGBDs paralelos contraindicados para operações de larga escala, já que em caso de falhas todas as operações de processamento precisam ser refeitas. Com esta diferença fundamental, são apresentados a seguir prós e contras do modelo *MapReduce*.

Prós.

O modelo *MapReduce* é simples e eficiente para computar agregações. Devido a isto, é muitas vezes comparado com a consulta de processamento "*filtering then group-by aggregation*" dos SGBDs. Listamos aqui as principais vantagens em utilizar o modelo *MapReduce* para processamento de dados.

- *Tolerante à falhas. MapReduce* é altamente tolerante à falhas. Por exemplo, em [21] relata-se que a implementação *MapReduce* pode continuar o processamento correto dos dados apesar de uma média de 1,2 falhas por trabalho de análise.

- *Flexível.* O *MapReduce* não possui qualquer dependência de modelo de dados e esquema. Com este modelo de programação, um programador pode lidar com dados irregulares e desestruturados mais facilmente do que a partir de um SGBD.

- *Independente de armazenamento.* O *MapReduce* é basicamente independente de camadas de armazenamento subjacentes. Isto significar que o *MapReduce* pode trabalhar com diferentes tipos de camadas de armazenamento, tais como HBase [37], Bigtable [38], e outras [39].

- *Simples e fácil de usar.* O modelo *MapReduce* é simples, mas expressivo. Com *MapReduce*, um programador define seu trabalho unicamente com funções *Map* e *Reduce*, sem precisar especificar distribuições físicas deste trabalho através dos nós.

- *Alta Escalabilidade.* A melhor vantagem para utilizar o modelo *MapReduce* é sua alta escalabilidade. A empresa Yahoo! reportou que a utilização da implementação Hadoop escalou acima de 4.000 nós em 2008 [40].

Contras.

Apesar de várias vantagens, o modelo *MapReduce* carece de algumas funcionalidades que são comprovadamente primordiais para análise de dados em SGBDs. Neste contexto, *MapReduce* é comumente caracterizado como uma ferramenta ETL (*Extract-Transform-Load*). Dentre as desvantagens do framework *MapReduce* em comparação a SGBDs, têm-se:

- *Não há linguagem de alto nível.* O modelo *MapReduce* por si só não suporta qualquer linguagem semelhante à SQL nos SGBDs e qualquer técnica de otimização. Usuários devem codificar suas operações nas funções *Map* e *Reduce*.

- *Não usa esquemas e índices.* O *MapReduce* carece de esquemas e índices. Um *job MapReduce* pode funcionar logo após sua entrada na camada de armazenamento. Entretanto, este processamento improvisado descarta os benefícios da modelagem de dados. O *MapReduce* requer a conversão de cada item na leitura das entradas em objetos de dados para o processamento destes dados, causando degradação na performance.

- *Um único fluxo de dados fixo. MapReduce* fornece a facilidade de uso com uma simples abstração, mas em um fluxo de dados fixo. Sendo assim, diversos algoritmos complexos são difíceis de implementar utilizando apenas funções *Map* e *Reduce* em um único *job MapReduce*. Além disso, alguns algoritmos que requerem

múltiplas entradas não são bem suportados desde que o fluxo de dados do *MapReduce* é originalmente desenhado para ler um único conjunto de dados de entrada e gerar um único conjunto de dados de saída.

– *Baixa eficiência.* Sendo a tolerância à falhas e a escalabilidade os principais objetivos do *MapReduce*, suas operações não são otimizadas para eficiência de entrada e saída. As operações *MapReduce* são bloqueantes, ou seja, uma transição para o próximo estágio não pode ocorrer até que uma operação anterior esteja completa. Consequentemente, o paralelismo talvez não seja explorado de maneira eficaz. Além disso, todas as entradas de dados para um job *MapReduce* devem ser preparadas para o processamento (conversão de dados puros em objetos de dados).

A Tabela 1 sumariza os itens de vantagens e desvantagens na utilização do *MapReduce* como forma de processar dados em um sistema qualquer, realizando a correlação entre estes itens. Algumas destas desvantagens tem sido objeto de estudo de vários trabalhos. Por exemplo, no Twister, é introduzido o suporte de loops de execução [72] e, no Dryad, é proposto um esquema de grafos de execução para o *MapReduce* [73].

Vantagens	**Desvantagens**
Tolerante à falhas	Baixa eficiência
Flexível	Não há linguagem de alto nível
Independente de armazenamento	Não usa esquemas e índices
Simples e fácil de usar	Um único fluxo de dados fixo
Alta escalabilidade	Não usa esquemas e índices

Tabela 1. Sumário com vantagens e desvantagens do modelo *MapReduce* em relação à SGBDs.

3 Arquiteturas e Implementações do *MapReduce*

O *MapReduce*, assim como sua implementação *open-source*, Hadoop, objetiva o paralelismo na computação em grandes clusters. Entretanto, outras implementações têm sido propostas para diferentes tipos de ambientes, tais como: Phoenix [41], que implementa o modelo *MapReduce* para sistemas de memória compartilhada, e a Mars [42], a qual implementa *MapReduce* para processadores gráficos. Nesta seção, iremos apresentar as principais implementações *MapReduce*, focando naquelas que envolvem paradigmas da computação paralela.

3.1 *MapReduce* Padrão

A implementação original do *MapReduce* foi realizada pelo Google, tendo como alvo grandes clusters de máquinas em rede [21]. A biblioteca *MapReduce* do Google lida de forma nativa com o paralelismo e a distribuição de aplicações, permitindo que

desenvolvedores não precisem se preocupar com estas questões, focando apenas no problema real, *i.e.* implementar funções *Map* e *Reduce*. Outra implementação do modelo padrão *MapReduce* é a Hadoop, uma biblioteca *open-source* desenvolvida pela Apache [9] cuja arquitetura é, essencialmente, a mesma definida na implementação original do Google.

Nestas implementações, os dados de entrada e saída são distribuídos e salvos em discos locais de máquinas conectadas em rede. Para gerenciar estes dados, ambas implementações possuem sistemas gerenciadores de arquivos. A implementação *MapReduce* do Google utiliza o Google File System (GFS), um sistema de arquivos distribuídos utilizado para supervisionar os dados armazenados ao longo do *cluster*. Por outro lado, a implementação Hadoop distribui e armazena os dados utilizando o Hadoop Distributed File System (HDFS), que também possui a característica de replicar blocos de dados para aumentar a confiabilidade. Sendo assim, estas implementações, em essência, são compostas por duas partes principais: o elemento *MapReduce*, para o processamento distribuído, e o sistema de armazenamento.

Apesar de o modelo original proposto pelo Google, a implementação do *MapReduce* pelo projeto da Apache, Hadoop, é, sem dúvidas, a mais popular variante atualmente utilizada em um número crescente de companhias e projetos científicos proeminentes [43]. Neste contexto, diversas propostas de implementações *MapReduce* e aplicações que utilizam processamento distribuído fazem uso do Hadoop como solução subjacente, principalmente devido à sua propriedade *open-source*. Além dos componentes principais destas implementações, tanto o *MapReduce* original quanto o Hadoop possuem outras bibliotecas e subsistemas que os compõem, mas que não entramos em detalhes devido ao espaço textual.

3.2 Processadores Multicore

Dentre dos esforços de otimizar ou implementar o modelo *MapReduce* através do uso de processadores *multicore*, se destacam os trabalhos Phoenix [41] [45] [46], MJ-R [47], MATE [48] [49] [50] e Tiled-*MapReduce* [51]. Essas pesquisas buscam melhorar a produtividade no desenvolvimento de códigos paralelos para arquiteturas *multicore* através do uso do modelo de programação *MapReduce*, bem como o desempenho resultante de sua utilização.

O Phoenix [45] é uma implementação do *MapReduce* criada em Stanford voltada para chips *multicore* e multiprocessadores de memória compartilhada. O Phoenix inclui uma API de programação, uma biblioteca desenvolvida em C e C++, e um sistema de *runtime* que gerencia automaticamente a criação de *threads*, o escalonamento dinâmico de tarefas, a partição de dados e o processo de tolerância a falhas.

A implementação do Phoenix é baseada tanto no uso de *threads workers* para executar tarefas paralelas, sendo criada uma *thread* para cada núcleo ou thread de hardware disponível, e quanto no uso de *buffers* em memória compartilhada, para facilitar a comunicação sem a cópia excessiva de dados. O Phoenix introduz duas funções adicionais, *Split* e *Partition*, para fazer a divisão dos dados de entrada para as tarefas *Map* e *Reduce*. A função *Split* é chamada durante o estágio de *Map* tanto para particionar os dados de entrada em pedaços que caibam nas memórias cache L1 dis-

poníveis, quanto indicar um ponteiro para os dados da tarefa a ser executada. A tarefa *Map* aloca o número de *workers* necessários para a execução da função *Map*, então cada *worker* processa os dados e emite um par de chaves intermediárias.

Por sua vez, a função *Partition* separa os pares de chave/valor intermediários em unidades distintas para a execução das tarefas *Reduce*. A função assegura que todos os valores da mesma chave estarão na mesma unidade. Dentro de cada *buffer* os valores são ordenados por chave através de um algoritmo de ordenação. O escalonador aguarda todas as tarefas *Map* encerrarem para depois lançar as tarefas *Reduce*. A tarefa *Reduce* aloca os *workers* necessários para processarem as tarefas de maneira dinâmica. Uma mesma chave é agrupada em uma única tarefa para ser repassada para a tarefa *Reduce* correspondente. Por causa disso, pode-se ter grande desbalanceamento de carga. Assim, o escalonamento dinâmico é muito importante para evitar este problema. Quando as saídas *Reduce* estão ordenadas por chave, é executado um *merge* de todas as tarefas em um único *buffer*.

Os pares intermediários gerados pelas tarefas *Map* são armazenados em uma matriz de quantidade fixa de colunas e uma linha para cada partição de dados gerada pela função *split*. Cada linha atua como uma tabela *hash* com uma entrada por coluna. Contudo, um par é armazenado em uma coluna determinada pelo *hash* da chave do par. Como consequência, cada tarefa *Reduce* do Phoenix processa uma coluna inteira. Esta configuração limita a contenção no sentido que diferentes *threads Map* podem escrever seus resultados intermediários em diferentes áreas de memória.

Yoo *et al* [45] e Talbot *et al* [46] propuseram otimizações para o Phoenix. Apesar da versão original Phoenix ter apresentado desempenho aceitável em processadores *multicore* de escala pequena com latência de acesso uniforme de memória (UMA), os autores constaram que o Phoenix apresenta desempenho significativamente inferior em processadores de maior escala com latência de acesso não uniforme de memória (NUMA). A perda de desempenho se deve ao aumento da latência de memória, problemas de localidade e alta contenção quando são usadas *threads* espalhadas por múltiplos chips.

Uma das otimizações propostas foi introduzir uma fila de tarefas por grupo de localidade e distribuir as tarefas de acordo com a localidade pertencente ao pedaço de dados a ser processado [45]. As *threads workers* selecionam as tarefas a serem processadas das suas filas locais de tarefas e, quando suas filas estiverem vazia, poderão "roubar" tarefas de filas de outras *threads*. Outra otimização foi a execução *prefetching* de dados que serão utilizados no futuro para diminuir a latência média de acesso [45]. Por fim, Yoo *et al* observaram que o tipo de chamada de sistema usada para alocar e deslocar memória influencia significativamente no desempenho [45]. Se as chamadas de sistema possuem problemas de escalabilidade, o desempenho resultante da aplicação será deteriorado com o aumento de *threads* ou dados. Os supracitados autores avaliaram através de perfiladores de código diferentes funções de manipulação de memória, mas não conseguiram encontrar uma implementação que resultasse em um bom desempenho para todos os cenários.

No mais, Phoenix original apresentava um *pipeline* fixo dificultando, ou até impedindo, que os desenvolvedores conseguissem adaptá-lo para características específicas do *workload* de dados a ser processado. Talbot *et al* [46] propõem uma implementa-

ção alternativa, chamada Phoenix++, que provê uma abstração mais flexível de armazenamento dos pares intermediários na qual os usuários podem adaptá-la para condições específicas do *workload* a ser executado. Outra otimização imposta é o uso de uma função de combinação de dados que minimiza o uso de memória no sentido que reduz a quantidade de dados mantida ao longo de todo o ciclo de vida do processamento.

Um trabalho concorrente ao Phoenix é o MR-J [47], que é um *framework MapReduce* Java voltado também para ambientes *multicore*. O MR-J possui uma API muito similar ao do Hadoop que somente requer a implementação das funções *Map* e *Reduce*, diferentemente do Phoenix que exige a implementação adicional das funções *split* e *partition*. Contudo, o fluxo de execução é baseado no Phoenix. Os dados de entrada são particionados em unidades menores de tamanhos idênticos e serão processadas por sub-tarefas alocadas as *threads workers*.

A característica mais marcante do MR-J é uso do *framework* ForkJoin [52], presente a partir do Java 7. As *threads workers* são do tipo ForkJoinTask e permitem um método mais eficiente de processamento paralelo ao dividir um problema em tarefas menores e executá-las independentemente, sem uso de sincronização. A criação das *threads* é realizada através de um *pool* ForkJoin. Outra funcionalidade presente é o uso de encadeamento de *jobs* cujo objetivo é reaproveitar *threads* e estruturas de dados entre jobs de *MapReduce* distintos.

Outra solução *MapReduce* voltada para processadores *multicore* é o MATE [50] que foi desenvolvido através do *framework* Phoenix. No MATE, as tradicionais funções *Map* e *Reduce* são substituídas por uma única função *reduction* que atualiza objetos de dados (também chamados de *reduction*). Ao invés de fazer a emissão de pares chave/valor, a atualização dos objetos reduz a quantidade de pares gerados para uma mesma chave. Apesar desta abordagem aumentar a possibilidade de condições de corrida, isso é evitado através da replicação dos objetos *reduction* de tal forma que cada *thread* possui seu objeto. Ao final, uma função de combinação é usada para computar os dados das réplicas de objetos *reduction*, emitindo, assim, o resultado final. O restante do funcionamento segue o mesmo princípio do Phoenix.

Por sua vez, o EX-MATE [49] é uma extensão que introduz o suporte do armazenamento em disco de objetos *reduction* de escala maior. No MATE original, as funções *reduction* somente processavam objetos *reduction* que estivessem integralmente mantidos na memória, consequentemente, limitando a quantidade de aplicações que poderiam ser implementadas. A alternativa proposta foi o particionamento de objetos *reductions* e o carregamento de apenas um subconjunto na memória. O particionamento e o carregamento parcial de dados evitam a execução frequente de operações de acessos a disco bem como viabiliza a operação com conjunto maiores de dados. Ademais, com objetivo de otimizar as leituras com base em padrões de acesso aos dados, o MATE permite a especificação do algoritmo de particionamento a ser usado.

Por fim, além dos trabalhos acima mencionados, há diversos trabalhos que objetivam otimizar o funcionamento das soluções *MapReduce* para *clusters* através do uso de CPU multicores. O Tiled-*Map-Reduce* (TMR) [51] argumenta que várias das soluções existentes não aproveitam o potencial dos processadores multicores existentes em clusters. O TMR propõe a utilização da estratégia *tiling*, também conhecida como

blocagem, que consiste na ideia de agrupar dados (e consequentemente *threads*) em pequenos blocos de forma a reduzir a quantidade de acessos à memória global através da utilização de um mecanismo de *caching* compartilhado entre essas *threads*. Nesse sentido, o TMR combina tarefas de *map* e *reduce* em grupos pequenos com objetivo de aumentar a localidade dos dados e iterativamente faz o processamento destes grupos em nós específicos do *cluster*. Esta técnica permitiu um *speedup* máximo de 3,5 em relação ao Phoenix.

Além dos trabalhos apresentados acima, ainda há outros trabalhos voltados a implementação do modelo *MapReduce* para outros tipos de processadores. Os trabalhos [65, 66] apresentam propostas de implementação do *MapReduce* para processadores *Cell Broadband Engine* (*Cell BE*) da IBM.

3.3 Processadores GPU

Diversos trabalhos propõem a implementação do *MapReduce* para o uso específico de GPUs ou a otimização do modelo de *cluster* tradicional para executar de forma mais eficiente computações vetoriais ou matemáticas. Dentre estes trabalhos, destaca-se o Mars que propõe um framework *MapReduce* para processadores GPU da NVIDIA que pode ser utilizado para máquina de memória compartilhada ou de memória distribuída [42].

Similar ao *MapReduce* original, o Mars tem duas fases, *map* e *reduce*. Na etapa *Map*, a função *split* divide os dados de entrada em pedaços de igual ao número de *threads* a serem criadas. A quantidade de *threads* a serem criadas é determinada de acordo à ocupação da GPU, contudo essa configuração pode também ser especificada pelo programador.

Como as GPUs atuais não permitem a alocação dinâmica de memória, o Mars realiza um pré-processamento das tarefas *map* e *reduce* a fim de calcular a quantidade de dados a serem emitidos para alocar a memória necessária dentro das GPUs. Uma vez estimado o tamanho dos dados, são alocados *arrays* de tamanho fixo na memória da GPU. No mais, o Mars processa em CPU a entrada de dados proveniente de arquivos armazenados discos rígidos, transformando-os em pares chave/valor na memória principal, para ser transferida para a GPU.

Contudo, a quantidade de dados a serem geradas pelas tarefas *map* e *reduce* são desconhecidas, inviabilizando a alocação prévia de *arrays* para armazenamento dos dados. O Mars propõe um esquema de emissão de resultados em dois passos. O primeiro passo calcula o tamanho dos resultados gerados para cada tarefa. O segundo passo emite o resultado na memória do dispositivo. O cálculo da quantidade de dados a serem armazenadas é efetuado através do computo da soma dos prefixos utilizados pelas chaves a serem utilizadas.

Outras otimizações suportadas pelo Mars é a leitura de vetores de dados através de uma única instrução. A leitura de dados agrupados (*coalesced*) também permite aprimorar a performance de acesso a memória no sentido em que um grupo de dados é carregado na memória, diminuindo, desta forma, a execução de operações de requisição de dados. Esta funcionalidade se torna ainda mais útil quando um grupo de *threads* compartilha dados entre si.

Outra implementação de *MapReduce* que explora o uso de GPU é o MapCG[64]. O fluxo de execução do MapCG é similar ao Mars, sendo as principais contribuições o suporte a portabilidade de código entre CPU e GPUs e a operações atômicas no gerenciamento de memória compartilhada. No que tange as operações atômicas, estas são usadas para a escrita segura dos dados intermediários gerados pelas tarefas *map* e do resultado final gerado pelas tarefas *reduce*. O uso de operações atômicas evita o uso do esquema de escrita em duas fases usado pelo Mars e oferece um melhor desempenho para aplicação. No mais, o MapCG usa tabelas *hashs* para agrupar os dados intermediários gerados pelas funções *Map*. O uso de tabelas *hash* resultou em uma ordenação dos dados mais eficiente do que a apresentada no Mars.

Contudo, o uso de tabelas *hash* introduz um gargalo gerado pelas sucessivas alocação e desalocação de memória. Este mesmo problema foi primeiramente relatado no trabalho Phoenix 2 [45]. O MapCG propõe o uso de um alocador especializado de memória baseado em premissas de processamento paralelo. O alocador é otimizado para lidar com grandes quantidades de pequenos blocos de dados e desalocação de memória em grupo. Cada thread possui um *buffer* de memória que será usada para alocar seus blocos de memória. Quando uma *thread* invocar uma instrução da alocação, esta verificará se há disponibilidade em seu *buffer*. Caso haja disponibilidade, não é necessário executar fazer uma chamada de sistema para alocação de memória, sendo todo o processo executado em nível de usuário. Caso não tenha espaço disponível no *buffer,* é realizada uma chamada de sistema para alocação de um novo bloco de memória com o mesmo tamanho do *buffer* atual. Este novo bloco é adicionado à lista de blocos do *buffer*, assim dobrando o tamanho do *buffer*.

O alocador de memória possui uma adaptação quando executados em GPUs. Como as GPUs não possuem suporte a alocação dinâmica de memória, um grande *buffer* compartilhado é criado para todas as *threads*. Neste caso, o alocador ao receber uma invocação de alocação efetuará apenas um incremento nos ponteiros de blocos utilizados, desta forma não utilizando chamadas de sistema para alocação de memória. Independente da implementação do alocador de memória, todas as operações alocação e desalocação de memórias são executadas de forma atômica.

Outra implementação do *MapReduce* que suporta execução em GPU é o MATE-CG [32] que é uma extensão do MATE [31]. O MATE-CG possui três modos de execução, dois permitem o processamento apenas em CPU ou GPU e outro modo que efetua o processamento em ambos os tipos de processador. Nos três casos, a execução é distribuída entre nós de um *cluster*. Contudo, o MATE-CG suporta apenas GPUs da fabricante NVIDIA. O MATE-CG possui uma *runtime* responsável por particionar os dados entre os nós existentes do *cluster* e posteriormente dividir a carga alocada de cada nó com os núcleos da CPU e da GPU existentes. Esta mesma *runtime* é responsável por controlar a comunicação entre um nó e seus núcleos.

A principal alteração em relação à versão original do MATE é a introdução de duas novas funções a serem implementadas pelo usuário: *cpu_reduction* e *gpu_reduction*. Ambas as funções são responsáveis por processar os dados de entrada e atualizar os objetos *reduction* . Contudo, a segunda contém código escrito através do *framework* CUDA e precisa lidar com a transferência de dados entre a memória RAM convencional e memória da GPU.

Outra funcionalidade importante é a definição do tamanho das fatias de dados que serão alocadas para cada núcleo de CPU e GPU. A *runtime* provê um valor *default* com base em um modelo analítico e aprendizagem de máquina não incremental (*off-line*). Este valor *default* é definido com escalas de 16 MB de dados. Esta configuração também pode ser definida no projeto da aplicação pelo desenvolvedor.

O uso de GPU pode melhorar significativamente certas de classes de computação [32, 42]. Contudo, o modelo de programação não é trivial, ainda exibe dificuldades de programação como a transferência de dados entre as memórias. Em ambientes heterogêneos a escolha do mapeamento de dados adequado entre a CPU e GPU é crucial para o desempenho geral das aplicações. No mais, além dos trabalhos acima apresentados que propõem implementações do *MapReduce* específicas para GPUs da NVIDIA, há trabalhos que exploram GPUs de outros fabricantes, tais como [67].

3.4 Outras Arquiteturas de Processamento

Como já mencionado, diversas implementações do *MapReduce* foram propostas para várias arquiteturas computacionais. A vasta maioria procura otimizar as versões voltadas para clusters, ou incorporar o uso de processadores multicore ou GPUs. Ainda, mais recentemente, foram apresentados modelos que exploram outras arquiteturas computacionais de memória distribuída além de clusters.

O trabalho P2P-*MapReduce* é um framework que propõe o uso de *MapReduce* em infraestruturas em nuvem [68]. Contudo, o mesmo procura explorar o modelo *peer-to-peer* para execução de tarefas computacionais em ambientes amplamente distribuídos, autônomo e heterogêneo. Neste modelo, todos os nós podem atuar tanto como *worker*, que executa tarefas computacionais, quanto como *master*, que coordena fluxo de execução. Os nós *master* replicam seus dados em outros nós *master* para permitir a recuperação de um trabalho de computação em caso de falha. As falhas de nós *workers* são manipuladas através da simples replicação das tarefas enviadas para múltiplos nós. Desta forma, o P2P-*MapReduce* gerencia o participação intermitente dos nós, falhas dos nós *master* e recuperação descentralizada de trabalhos. Diferentemente do Hadoop que utiliza um sistema de arquivo distribuído para realizar a transferência dos dados utilizados e produzidos ao longo do ciclo de vida, o P2P-*MapReduce* transferem os dados entre os nós participantes diretamente através os protocolos FTP e HTTP.

Há também trabalhos, a exemplo do MOON[69] e do Tang et al.[70], que procuram explorar a capacidade de processamento não utilizada pelos computadores através de computação oportunística e voluntária. Tipicamente, sistemas de computação voluntária reúnem um grupo significativo de computadores não confiáveis e voláteis que podem entrar e sair do sistema a qualquer momento fazendo com que dados e trabalho executado sejam perdidos.

O MOON é um sistema de computação oportunística que estende o Hadoop introduzindo uma arquitetura híbrida na qual os nós *workers* são classificados de acordo com a sua confiabilidade [69]. Os nós confiáveis são chamados de dedicados armazenam arquivos que não podem ser perdidos sob nenhuma circunstância. Por sua vez, os nós voláteis armazenam os arquivos de dados transientes que podem suportar um

certo grau de indisponibilidade. Independentemente do tipo de dado, um arquivo é replicado em mais de um nó. Como, intuitivamente, os nós dedicados possuem maior disponibilidade que os nós voláteis, este esquema de alocação e replicação pode diminuir de forma significativa a sobrecarga de replicação total do sistema. Além disso, os nós dedicados podem ser usados para executarem tarefas de longa duração, aumentando, assim, a probabilidade das tarefas serem completadas.

Outra contribuição do MOON é a criação de um estado de hibernação para quando um nó se tornar indisponível. Em sistemas de computação oportunística, os donos dos computadores sempre têm o controle sobre seus recursos. Quando o dono volta a operar sua máquina, que estava ociosa, as tarefas que estavam oportunisticamente sendo executadas são suspensas. O MOON para lidar com esse comportamento aguarda por um determinado tempo para verificar se o nó voltou a processar a tarefa ou não. Isto evita o relançamento de tarefas em situações nas quais os recursos ficam indisponíveis por um curto período de tempo.

Tang et al. [70] apresenta um sistema similar ao MOON que propõe o uso do *middleware* BitDew para execução de tarefas *MapReduce* em grades computacionais de desktops. Assim como no MOON, a arquitetura proposta classifica os nós quanto a sua confiabilidade. Nós voláteis assumem o papel de *worker*, e os nós estáveis assumem os papéis de *master* ou *service*. O nó *service* é responsável pelo monitoramento das tarefas do *MapReduce*, e é controlado pelo nó *mestre*. Para lidar com a volatilidade dos recursos, a solução utiliza os mecanismos de tolerância a falhas do BitDew.

Seguindo a mesma linha de computação oportunística, ainda há trabalhos que propõem a construção de uma grade computacional através da agregação de dispositivos moveis. O Misco é um destes trabalhos e apresenta um framework de *MapReduce* desenvolvido para dispositivos móveis [71]. O Misco busca otimizar os recursos de dispositivos móveis de forma a aproveitar a capacidade ociosa, administrando a baixa conectividade e disponibilizando serviços ágeis para diversas localizações geográfica.

Apesar do Misco seguir projeto básico do *MapReduce*, ele varia em dois aspectos: atribuição de tarefas e transferência de dados. O primeiro aspecto é gerenciado através do uso de uma estratégia de *polling*, na qual os nós *workers* notificam o nó *master* sempre que se tornam disponíveis. Se não há tarefas para executar, o nós *worker* esperam um determinado período de tempo para requisitar uma nova tarefa novamente. No que tange a transferência de dados, ao invés de utilizar um sistema de arquivo distribuído, que evidentemente não é viável para o cenário de dispositivos moveis, o Misco usa o protocolo HTTP a fim de enviar requisições, sincronizar informações sobre as tarefas e transferir os dados de uma forma geral.

4 Aplicações

Nesta seção, serão descritas algumas aplicações que utilizam *MapReduce* em diferentes contextos para atingir seus objetivos, visando demonstrar a flexibilidade relativa a este modelo de programação e processamento de dados. Uma destas aplicações é a Mahout, um projeto da Apache que visa construir bibliotecas de aprendizagem de

máquina escaláveis, as quais são executadas em paralelo em virtude do uso subjacente do Hadoop [53]. Os projetos RHIPE e Ricardo são ferramentas que integram a ferramenta estatística R e o Hadoop para suportar análise paralela de dados [54] [55].

Aplicações de tempo real e de *streaming* de dados também possuem soluções baseadas em *MapReduce*, como a proposta do Facebook para oferecer suporte ao *MapReduce* para lidar com análise de dados em tempo real [56]. O *MapReduce* é utilizado também em aplicações de bioinformática, a exemplo do Cloudburst [57] que é utilizado para mapear sequenciamentos do genoma, utiliza Hadoop para processar os dados obtidos.

Outros exemplos de aplicações, alguns identificados por [58], são: o HIPI (utilizado para conversão massiva de imagens) [59], o Hadoop-GIS (sistema escalável e de alta performance para executar consultas espaciais de larga escalas utilizando o Hadoop) [60], a solução da IBM para buscas analíticas [61], dentre outras. É importante ressaltar a utilização da implementação Hadoop como plataforma subjacente. Conforme já mencionamos, isso se deve à tanto singularidade *open-source* quanto maturidade da solução.

5 Discussão

O *MapReduce* está se tornando ubíquo, ainda que sua eficiência e performance sejam controversas. Não existe nenhuma novidade sobre os princípios utilizados no *MapReduce* [31] [33]. Entretanto, o *MapReduce* mostra que vários problemas podem ser resolvidos com sua utilização, de forma única, sem precedentes. Devido aos frequentes *checkpoints* e a programação especulativa do tempo de execução, o *MapReduce* apresenta-se com baixa eficiência. Ainda assim, o modelo de programação presente no *MapReduce* se torna necessário para atingir uma alta escalabilidade e tolerância a falhas no processamento de dados massivo. Então, aumentar a eficiência e garantir o mesmo nível de escalabilidade junto com a tolerância a falhas é o maior desafio das propostas e implementações *MapReduce*. Espera-se resolver o problema da eficiência de duas formas: (1) melhorando o próprio modelo *MapReduce*, como foi visto em algumas implementações focadas em um domínio específico, e (2) alavancando a infraestrutura de *hardware* subjacente.

A resposta para "como utilizar as funcionalidades de *hardware* moderno" ainda não foi obtida em diversas áreas. Entretanto, dispositivos de computação modernos tais como: multiprocessadores no nível de *chip* e Solid State Disk (SSD) podem ajudar a reduzir o tempo de computação e as operações de entrada e saída no *MapReduce* significativamente. O uso de SSD com Hadoop foi brevemente analisado em [62]. A autoconfiguração e a programação de *jobs* para execução em ambientes multiusuários também são questões que ainda não foram bem resolvidas, como utilizar técnicas da computação paralela para diminuir a contenção entre os nós dependentes.

6 Conclusão

No presente trabalho, discutiu-se as vantagens e desvantagens do *MapReduce* (e demais implementações ou extensões), focando em soluções voltadas para utilização de computação paralela no processamento dos dados. Para fins de comparações, utilizou-se como parâmetro o processamento de dados realizado por SGBDs, mostrando como estas duas ferramentas podem se complementar. Em algumas soluções discutidas, percebeu-se como alguns paradigmas da computação paralela podem aumentar a eficiência do modelo *MapReduce*, diminuindo a incidência de operações de entrada e saída, e a contenção no processamento dos dados.

Referências

[1] T. Takahashi, Sociedade da informação no Brasil: livro verde., Ministério da Ciência e Tecnologia (MCT), 2000.

[2] Anderson, David P., and Gilles Fedak. "The computational and storage potential of volunteer computing." *Cluster* Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on. Vol. 1. IEEE..

[3] Andrade, Nazareno, et al. "OurGrid: An approach to easily assemble grids with equitable resource sharing." Job scheduling strategies for parallel processing. Springer Berlin Heidelberg, 2003..

[4] Zhang, Nan, Yun-shan Chen, and Jian-Li Wang. "Image parallel processing based on GPU." Advanced Computer Control (ICACC), 2010 2nd International Conference on. Vol. 3. IEEE, 2010..

[5] Kurzak, Jakub, et al. "The playstation 3 for high performance scientific computing." (2008)..

[6] Ekanayake, Jaliya, and Geoffrey Fox. "High performance parallel computing with clouds and cloud technologies." Cloud Computing. Springer Berlin Heidelberg, 2010. 20-38..

[7] Yang, Hung-chih, et al. "*Map-reduce*-merge: simplified relational data processing on large clusters." Proceedings of the 2007 ACM SIGMOD international conference on Management of data. ACM, 2007..

[8] *MapReduce* in Wikipedia, http://en.wikipedia.org/wiki/*MapReduce* (accessed August 2014).

[9] Hadoop, http://hadoop.apache.org (accessed August 2014).

[10] Qizmt, http://code.google.com/p/qizmt/ (accessed August 2014).

[11] MapSharp, http://mapsharp.codeplex.com/ (accessed August 2014).

[12] Elastic, https://github.com/tc/elastic-*MapReduce*-ruby (accessed August 2014).

[13] MrJob, http://pythonhosted.org/mrjob/ (accessed August 2014).

[14] Octopy, http://code.google.com/p/octopy/ (accessed August 2014).

[15] Hadoop Wiki, https://wiki.apache.org/hadoop/PoweredBy (accessed August 2014).

[16] Jiang, David, A. Tung, and Gang Chen. "*MAP*-JOIN-*REDUCE*: Toward scalable and efficient data analysis on large clusters." Knowledge and Data Engineering, IEEE Transactions on 23.9 (2011): 1299-1311..

[17] Verma, Abhishek, et al. "Scaling genetic algorithms using *MapReduce*." Intelligent Systems Design and Applications, 2009. ISDA'09. Ninth International Conference on. IEEE, 2009..

[18] Zaharia, Matei, et al. "Improving *MapReduce* Performance in Heterogeneous Environments." OSDI. Vol. 8. No. 4. 2008..

[19] Lang, Willis, and Jignesh M. Patel. "Energy management for *MapReduce* clusters." Proceedings of the VLDB Endowment 3.1-2 (2010): 129-139..

[20] Kasim, Henry, et al. "Survey on parallel programming model." Network and Parallel Computing. Springer Berlin Heidelberg, 2008. 266-275..

[21] J. Dean e S. Ghemawat, "*MapReduce*: simplified data processing on large clusters," *Magazine Communications of the ACM - 50th anniversary issue: 1958 - 2008,* pp. 107-113, January 2008.

[22] R. Lämmel, "Google's *MapReduce* programming model — Revisited," *Journal Science of Computer Programming,* pp. 208-237, October 2007.

[23] "T. White, Hadoop: The Definitive Guide. O'Reilly, 2009".

[24] "S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in Proceedings of the 19th ACM Symposium on Operating Systems Principles, 2003, pp. 29–43.".

[25] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz e I. Stoica, "Improving *MapReduce* Performance in Heterogeneous Environments," *OSDI'08 Proceedings of the 8th USENIX conference on Operating systems design and implementation,* pp. 29-42, 2008.

[26] "Kirpal A. Venkates, Kishorekumar Neelamegam, R. Revathy, Using *MapReduce* and load balancing on the cloud. IBM.," [Online]. Available: http://www.ibm.com/developerworks/cloud/library/cl-*MapReduce*/. [Acesso em 2014].

[27] C. Henderson, "Software Scalability with *MapReduce*," April 2010. [Online]. Available: http://craighenderson.co.uk/papers/software_scalability_*MapReduce*/. [Acesso em 2014].

[28] "M. Stonebraker and U. Cetintemel. One size fits all: An idea whose time has come and gone. In Proceedings of the 21st IEEE ICDE, pages 2–11. IEEE, 2005," [Online].

[29] "M. Stonebraker et al . One size fits all? Part 2: Benchmarking results. In Conference on Innovative Data Systems Research (CIDR), 2007.," [Online].

[30] “D. Florescu and D. Kossmann. Rethinking cost and performance of database systems. ACM SIGMOD Record, 38(1):43–48, 2009,” [Online].

[31] “D. DeWitt and M. Stonebraker. *MapReduce*: A major step backwards. The Database Column, 1, 2008.,” [Online].

[32] “A. Pavlo et al . A comparison of approaches to large-scale data analysis. In Proceedings of the ACM SIGMOD, pages 165–178, 2009.,” [Online].

[33] “M. Stonebraker et al. *MapReduce* and parallel DBMSs: friends or foes? Communications of the ACM, 53(1):64–71, 2010.,” [Online].

[34] “J. Dean et al . *MapReduce*: a flexible data processing tool. Communications of the ACM, 53(1):72–77, 2010.,” [Online].

[35] “A. Pavlo et al . A comparison of approaches to large-scale data analysis. In Proceedings of the ACM SIGMOD, pages 165–178, 2009.,” [Online].

[36] “E. Anderson et al . Efficiency matters! ACM SIGOPS Operating Systems Review, 44(1):40–45, 2010.,” [Online].

[37] “[Vora, M.N. ; Innovation Labs., Tata Consultancy Services (TCS) Ltd., Mumbai, India. - Hadoop-HBase for large-scale data Computer Science and Network Technology (ICCSNT), 2011 International Conference on (Volume:1) Date of Conference: 24-26 Dec. 2011,” [Online].

[38] “F. Chang et al . Bigtable: A distributed storage system for structured data. ACM Transactions on Computer Systems (TOCS), 26(2):1–26, 2008.,” [Online].

[39] “Azure Table Storage and Windows Azure SQL Database,” [Online]. Available: http://msdn.microsoft.com/en-us/library/azure/jj553018.aspx.

[40] “A. Anand. Scaling Hadoop to 4000 nodes at Yahoo! http://goo.gl/8dRMq, 2008.,” [Online].

[41] “Ranger, Colby, et al. "Evaluating *MapReduce* for multi-core and multiprocessor systems." High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on. IEEE, 2007.”.

[42] “He, Bingsheng, et al. "Mars: a *MapReduce* framework on graphics processors." Proceedings of the 17th international conference on Parallel architectures and compilation techniques. ACM, 2008.”.

[43] C. Doulkeridis e K. NØrvåg, “A survey of large-scale analytical query processing in *MapReduce*,” *The VLDB Journal — The International Journal on Very Large Data Bases,* pp. 335-380, June 2014.

[44] “Blake, Geoffrey, Ronald G. Dreslinski, and Trevor Mudge. "A survey of multicore processors." Signal Processing Magazine, IEEE 26.6 (2009): 26-37.”.

[45] “Yoo, Richard M., Anthony Romano, and Christos Kozyrakis. "Phoenix rebirth: Scalable *MapReduce* on a large-scale shared-memory system." Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on. IEEE, 2009.”.

[46] “Talbot, Justin, Richard M. Yoo, and Christos Kozyrakis. "Phoenix++: modular *MapReduce* for shared-memory systems." Proceedings of the second international workshop on *MapReduce* and its applications. ACM, 2011.”.

[47] “Kovoor, George. "MR-J: A *MapReduce* framework for multi-core architectures." Master's thesis, University of Manchester (2009).”.

[48] “Jiang, Wei, Vignesh T. Ravi, and Gagan Agrawal. "A *Map-Reduce* system with an alternate API for multi-core environments." Proceedings of the 2010 10th IEEE/ACM International Conference on *Cluster*, Cloud and Grid Computing. IEEE Computer Society, 2010.”.

[49] “Jiang, Wei, and Gagan Agrawal. "Ex-mate: data intensive computing with large reduction objects and its application to graph mining." Proceedings of the 2011 11th IEEE/ACM International Symposium on *Cluster*, Cloud and Grid Computing. IEEE Computer Society,”.

[50] “Jiang, Wei, and Gagan Agrawal. "Mate-cg: A *map reduce*-like framework for accelerating data-intensive computations on heterogeneous clusters." Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International. IEEE, 2012.”.

[51] “Chen, Rong, Haibo Chen, and Binyu Zang. "Tiled-*MapReduce*: optimizing resource usages of data-parallel applications on multicore with tiling." Proceedings of the 19th international conference on Parallel architectures and compilation techniques. ACM, 2010.”.

[52] “ForkJoim Framework,” [Online]. Available: http://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html.

[53] “Mahout: Scalable machine-learning and data-mining library. http://mapout.apache.org, 2010.,” [Online].

[54] “Saptarshi Guha. RHIPE- R and Hadoop Integrated Processing Environment. http://www.stat.purdue.edu/ sguha/rhipe/, 2010.,” [Online].

[55] “S. Das et al . Ricardo: integrating R and Hadoop. In Proceedings of the 2010 ACM SIGMOD, pages 987–998, 2010.,” [Online].

[56] “D. Borthakur, J. Gray, J. S. Sarma, et al. Apache Hadoop goes realtime at Facebook. In Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD), pages 1071-1080, 2011.,” [Online].

[57] “CloudBurst | CBCB,” [Online]. Available: www.cbcb.umd.edu/software/cloudburst. [Acesso em 2014].

[58] C. Lam, Hadoop in Action, Greenwich, CT: Manning Publications, 2010.

[59] L. L. S. A. J. L. C Sweeney, HIPI: A Hadoop Image Processing Interface for Image-Based *MapReduce* Tasks, S Thesis. University of Virginia, Department of Computer Science , 2011.

[60] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang e J. Saltz, Hadoop-GIS: A High Performance Spatial Data Warehousing System over *MapReduce*,

Volume 6 Issue 11: Proceedings of the VLDB Endowment, 2013.

[61] R. K. S. R. F. R. E. S. S. T. S. V. a. H. Z. Vuk Ercegovac, Building Analytics for Enterprise Search: IBM's Project ES2, Hadoop in Action, Manning Publications, 2010.

[62] B. Li et al . A Platform for Scalable One-Pass Analytics using *MapReduce*. In Proceedings of the 2011 ACM SIGMOD, 2011..

[63] Keckler, Stephen W., et al. "GPUs and the future of parallel computing." IEEE Micro 31.5 (2011): 7-17.

[64] Hong, Chuntao, et al. "MapCG: writing parallel program portable between CPU and GPU." Proceedings of the 19th international conference on Parallel architectures and compilation techniques. ACM, 2010.

[65] De Kruijf, Marc, and Karthikeyan Sankaralingam. "*MapReduce* for the cell broadband engine architecture." IBM Journal of Research and Development 53.5 (2009): 10-1.

[66] Rafique, M. Mustafa, et al. "CellMR: A framework for supporting *MapReduce* on asymmetric cell-based clusters." Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on. IEEE, 2009.

[67] Elteir, Marwa, et al. "StreamMR: an optimized *MapReduce* framework for AMD GPUs." Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on. IEEE, 2011.

[68] Marozzo, Fabrizio, Domenico Talia, and Paolo Trunfio. "P2P-*MapReduce*: Parallel data processing in dynamic Cloud environments." Journal of Computer and System Sciences 78.5 (2012): 1382-1402.

[69] Lin, Heshan, et al. "Moon: *MapReduce* on opportunistic environments." Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing. ACM, 2010.

[70] Tang, Bing, et al. "Towards *MapReduce* for desktop grid computing." P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2010 International Conference on. IEEE, 2010.

[71] Dou, Adam, et al. "Misco: a *MapReduce* framework for mobile systems." Proceedings of the 3rd international conference on pervasive technologies related to assistive environments. ACM, 2010.

[72] Ekanayake, Jaliya, et al. "Twister: a *runtime* for iterative *MapReduce*." Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing. ACM, 2010.

[73] Isard, Michael, et al. "Dryad: distributed data-parallel programs from sequential building blocks." ACM SIGOPS Operating Systems Review. Vol. 41. No. 3. ACM, 2007.