Implementação e Avaliação de Algoritmos de Ordenação Paralela em MapReduce

Cristina Duarte Murta, Mariane Raquel Silva Gonçalves, Paula de Morais Pinhão
Centro Federal de Educação Tecnológica de Minas Gerais (CEFET-MG)
Departamento de Computação
Belo Horizonte, Brasil
Email: cristina@decom.cefetmg.br

***Resumo*—Quantidades cada vez maiores de dados, conhecidas como Big Data, são um fato do mundo real e um desafio em termos de processamento de dados. Ordenação é uma das tarefas mais comuns em computação e ordenar grandes massas de dados é uma necessidade em vários processos. O modelo de programação paralela MapReduce tem sido amplamente adotado para processar dados em larga escala em agrupamentos de computadores. Apresentamos neste artigo a implementação de dois algoritmos paralelos de ordenação, o Quicksort Paralelo e o Ordenação por Amostragem. Ambos foram implementados no ambiente de programação MapReduce/Hadoop e testados quanto ao seu desempenho para ordenar dados distribuídos em várias máquinas. Uma variedade de experimentos revela o comportamento de ambos os algoritmos e indica que o algoritmo Ordenação por Amostragem apresenta melhor desempenho.**

***Keywords*-Ordenação paralela, computação paralela, MapReduce.**

I. Introdução

Na última década, a quantidade de dados gerada e processada por empresas e sistemas computacionais aumentou várias ordens de grandeza, tornando o processamento de dados um desafio para a computação sequencial. Uma estimativa da *International Data Corporation*(IDC) mostrou que o tamanho do universo digital alcançou 1,8 zettabytes (1ZB = 10^{21} bytes) em 2011, e a estimativa é dobrar a cada dois anos, chegando a 40 ZB em 2020 [1].

O processamento de quantidades grandes de dados requer o uso da computação paralela, atualmente feita em grandes aglomerados de computadores comuns [2], [3]. Um dos modelos de programação paralela mais populares atualmente é o MapReduce [4], que surgiu como uma alternativa aos modelos tradicionais de programação paralela, com o objetivo de simplificar a programação, permitindo que programador foque no desenvolvimento de sua tarefa, e não nos detalhes da paralelização da computação. O Hadoop é a implementação mais conhecida do MapReduce [5], [6].

Uma das tarefas essenciais da computação é a ordenação de dados. Estima-se que a ordenação seja responsável por parte considerável dos ciclos de processamento computacional [7]. Na ordenação paralela, fatores como a movimentação de dados, o balanço de carga, a latência de comunicação e a distribuição inicial das chaves influenciam significativamente o desempenho dos algoritmos [8]. E sabe-se, ainda, que inexiste um modelo teórico conhecido que possa ser aplicado para prever com precisão o desempenho de um algoritmo em diferentes arquiteturas [9]. A escolha de um algoritmo genérico para diferentes máquinas e instâncias do problema torna-se particularmente árdua, tendo em vista a variedade de algoritmos de ordenação paralela disponíveis na literatura, bem como as múltiplas alternativas para a composição de sistemas paralelos.

Portanto, estudos experimentais ganham importância na avaliação e seleção de algoritmos apropriados para ambientes paralelos. É preciso que estudos experimentais sejam realizados para que determinado algoritmo possa ser recomendado em certa arquitetura com alto grau de confiança. É nesse contexto que esse trabalho se justifica.

Este trabalho apresenta uma análise comparativa de desempenho dos algoritmos paralelos Ordenação por Amostragem e Quicksort Paralelo. Os algoritmos foram implementados no modelo MapReduce, em ambiente Hadoop. Vários aspectos do problema foram considerados, o que resultou em uma ampla gama de testes e resultados. Os resultados indicam que o desempenho do algoritmo Ordenação por Amostragem supera o desempenho do Quicksort Paralelo nas escalas testadas.

II. Contexto e Trabalhos Relacionados

Diferentemente dos algoritmos sequenciais, os algoritmos paralelos são muito dependentes da arquitetura paralela e do ambiente de execução. A história da computação paralela revela variados algoritmos de ordenação para um conjunto amplo de arquiteturas. Um estudo recente, com foco na arquitetura paralela obtida a partir de *clusters* de computadores comuns, aponta fatores que interferem no desempenho final da ordenação, dentre os quais ressaltamos [8]:

- Balanceamento de carga: a carga de dados distribuída para cada processador deve ser equilibrada, uma vez que o tempo de execução da aplicação é tipicamente limitado pelo tempo do processador mais sobrecarregado.
- Movimentação dos dados: uma vez que a quantidade de dados movimentada é um ponto crítico, a movimentação de dados entre processadores deve ser mínima durante a execução do algoritmo; caso contrário, o custo de troca de dados pode dominar o custo de execução e limitar a escalabilidade.
- Latência de comunicação: a latência de comunicação é definida como o tempo médio necessário para enviar uma mensagem de um processador a outro. A latência não representa grande impacto quando o

número de processadores é pequeno mas, para um maior número de máquinas, torna-se um fator importante. Quando a quantidade de dados a ser ordenada é pequena comparada ao número de processadores, a latência pode ter grande peso no tempo de execução. Assim, é fundamental reduzir o tempo de latência em sistemas distribuídos.
- Sobreposição de comunicação e computação: no processamento paralelo, a execução tem fases de computação e de comunicação entre os processadores. Deve-se procurar sobrepor as duas fases, executando-se tarefas de processamento e operações de entrada e saída ao mesmo tempo, o que permite melhor aproveitamento dos recursos.

Em linhas gerais, o fluxo de execução da ordenação paralela é composto pelas seguintes tarefas [8]: coleta de informações para distribuição dos dados nos processadores; em um processador é feito o cálculo do particionamento das chaves a partir das informações coletadas; transmissão da informação de particionamento aos processadores; movimentação dos dados de acordo com a divisão estipulada. O processamento local (em cada máquina) é realizado entre as etapas. Se necessário, o ciclo é repetido.

Há duas tarefas principais na comunicação: compor um vetor de divisão global e enviar os dados para os processadores adequados. A maioria dos algoritmos têm múltiplos estágios de computação local. Assim, pode ser muito vantajoso sobrepor este processamento local e a comunicação.

A. O Modelo de Programação Paralela MapReduce

Proposto em 2008, o modelo de programação paralela MapReduce é amplamente adotado nos ambientes acadêmico e empresarial [4]. O objetivo do modelo é simplificar a tarefa de programação em paralelo. Para isso, o modelo provê tolerância a falhas, distribuição de dados e balanceamento de carga, permitindo que o programador ocupe-se exclusivamente com o desenvolvimento da solução proposta [3].

A programação é baseada em duas funções principais, Map e Reduce. A função Map é aplicada aos dados de entrada e produz uma lista intermediária de pares <chave, valor>. Todos os valores intermediários associados a uma mesma chave são agrupados e enviados à função Reduce, que, por sua vez, combina esses valores para formar um conjunto sintético de resultados.

O Hadoop [5], [6] é uma das implementações mais conhecidas do modelo MapReduce. Ele provê o gerenciamento da computação distribuída de maneira escalável e confiável. Grandes instalações computacionais utilizam o ambiente Hadoop em seus *clusters*, para processar diariamente vários terabytes de dados [10]. O MapReduce/Hadoop é considerado atualmente a principal plataforma de programação paralela, por ser capaz de processar volumes de dados em escalas antes desconhecidas, usando computadores comuns [3].

III. Algoritmos para Ordenação Paralela

Essa seção apresenta os algoritmos focalizados neste trabalho – Ordenação por Amostragem e Quicksort Paralelo – e discute suas implementações no modelo MapReduce. Esses algoritmos estão descritos de forma esquemática em [8].

A. Quicksort Paralelo

O Quicksort é considerado o algoritmo de ordenação sequencial mais rápido para grande parte das ordenações [11]. A estratégia de dividir para conquistar utilizada pelo Quicksort pode ser naturalmente paralelizada, por isso o Quicksort é um dos algoritmos mais promissores para ordenação paralela [8].

A versão paralela do Quicksort utiliza pivôs para realizar o particionamento recursivo dos dados no conjunto de processadores. O algoritmo não necessita de sincronização, pois cada sublista gerada é associada a um único processo, que não precisa comunicar-se com os demais, uma vez que os dados são independentes.

O balanceamento da carga, fundamental para o bom desempenho do algoritmo, é gerenciado pelo processo de seleção do pivô [8]. O pivô é o elemento de referência para a divisão das partições. Na implementação em Hadoop, o primeiro passo do algoritmo é escolher o elemento pivô a partir dos arquivos de entrada. Na fase Map os arquivos de entrada são lidos e são formados os pares <chave, valor> para cada registro presente no arquivo. Comparados com o pivô, os dados são divididos em duas partições: conjunto de valores maiores que o pivô e conjunto de valores menores ou iguais ao pivô. Cada partição formada pode ser novamente dividida, com a escolha de um novo pivô e um novo particionamento dos dados. O número de partições no qual a entrada de dados deve ser dividida é um parâmetro determinado pelo usuário. O algoritmo realiza as divisões até atingir esse valor.

Realizada a divisão, cada partição é atribuída a um processador, que executa a tarefa Reduce. Na fase Reduce, cada processador ordena os dados localmente. Essa ordenação é realizada pelo próprio Hadoop. Após a ordenação local, cada arquivo contém um conjunto ordenado, e a concatenação de tais arquivos fornece o arquivo final ordenado.

A escolha do pivô é fundamental para manter o balanceamento das partições e, consequentemente, para o bom desempenho do algoritmo. Há diversas estratégias para a escolha do pivô, dentre elas a escolha aleatória, o primeiro dado, o último dado. Na presente implementação, optamos pela mediana entre n valores do conjunto de dados. Dessa forma, são escolhidos n valores aleatoriamente e a mediana entre eles é definida como o pivô.

A Figura 1 apresenta um diagrama da execução do algoritmo implementado em Hadoop. No desenho, há um único pivô que particiona os dados em duas partições. A partir dos dados de entrada, foi escolhido o pivô, no caso a letra 'E' (passo 2). Em seguida, o arquivo é processado pela função Map e são formados os pares <chave, valor> (passo 3). Após a leitura das chaves, o particionador divide

os dados em duas partições: uma com valores menores ou iguais ao pivô e outra com valores maiores que o pivô (passo 4). Essas partições são distribuídas nos nós do sistema, que executam a função Reduce (passo 5), realizando a ordenação das chaves (passo 6). Caso fosse determinado outro número de partições, os passos 2, 3 e 4 seriam repetidos, formando novas partições.

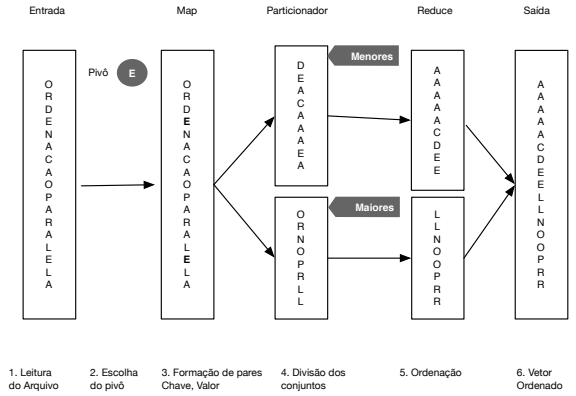


Figura 1. Fluxo de execução do algoritmo Quicksort Paralelo no modelo MapReduce

A implementação do Quicksort Paralelo, de acordo com o modelo MapReduce, é apresentada nos Algoritmos 1 e 2. O Algoritmo 1 apresenta o fluxo principal do programa, responsável por contabilizar as partições e definir o valor do pivô. Em seguida o algoritmo direciona a execução para as funções Map e Reduce, que realizam a divisão da entrada de dados.

Algorithm 1: Quicksort em MapReduce - Fluxo Principal

Entrada: Arquivos contendo chaves para ordenar, número de partições desejadas (P), endereço para escrita do arquivo ordenado
Saída : Arquivos ordenados

```
1  Fluxo Principal
2      numeroParticoes ← 1;
3      while numeroParticoes < P do
4          pivot = valor lido do arquivo ;
5          numeroParticoes = quicksort(); /* Funções
           Map, Reduce e Particionador */
6      end
7  end
```

O Algoritmo 2 descreve as funções Map, Reduce e o particionador. A função Map lê as chaves do arquivo de entrada e contabiliza o número de elementos maiores e menores ou iguais ao pivô. Nela é atribuído o número da partição como valor do par <chave, valor>. O particionador tem como função direcionar as chaves para as partições corretas. Ele recebe como entrada a chave, o valor e o número de partições definido, e retorna o valor, que representa o número da partição de destino. Cada partição é direcionada a uma função Reduce, que recebe chaves e valores e ordena localmente a subsequência.

Algorithm 2: Quicksort em MapReduce - Fases Map, Reduce e Particionador

```
1  Função Map(chave, valor)
2      Partição.Maiores ← identificador da partição com
       valores maiores;
3      Partição.Menores ← identificador da partição
       com valores menores;
4      if chave < pivo then
5          menores ← menores + 1 ;
6          Collect(chave, Partição.Menores) ;
7      else
8          maiores ← maiores + 1 ;
9          Collect(chave, Partição.Maiores) ;
10     end
11 end

12 Função Reduce(chave, Iterator<valores>)
13     foreach valor em valores do
14         Collect(chave,valor);
15     end
16 end

17 Função Particionador(chave, valor,
   númeroParticoes)
18     return valor;
19 end
```

B. Ordenação Por Amostragem

O algoritmo Ordenação por Amostragem é também um método de ordenação baseado na divisão do arquivo de entrada em subconjuntos. O algoritmo divide o arquivo de forma que as chaves de um subconjunto i sejam menores que as chaves do subconjunto $i+1$. Após a divisão, cada subconjunto é enviado a um processador, que ordena os dados localmente. Ao final, todos os subconjuntos são concatenados e formam um arquivo globalmente ordenado.

Nesse algoritmo, a etapa chave é dividir as partições de maneira balanceada, para que cada processador receba aproximadamente a mesma carga de dados. Para isso, é preciso determinar o número de elementos que devem ser destinados a uma certa partição. Isso é feito por amostragem das chaves do arquivo original, que permite estimar a distribuição dos valores das chaves a serem ordenadas.

A seleção das amostras pode ser feita de acordo com diferentes estratégias, cujo desempenho depende diretamente dos dados de entrada. O `RandomSampler` é considerado um bom amostrador de propósito geral [5] e foi o amostrador escolhido para nossa implementação do algoritmo Ordenação por Amostragem.

Na implementação feita em Java no ambiente Hadoop, as fases Map e Reduce foram implementadas como se segue. Uma amostra dos dados é coletada nos arquivos de entrada e armazenada em um vetor. Essa amostra

contém valores escolhidos aleatoriamente no conjunto de dados a ser ordenado. A partir do vetor de amostras, os dados são divididos em partições. O número de partições é determinado pelo número de máquinas e núcleos de processamento. Na fase Map os arquivos de entrada são lidos e são formados pares <chave, valor> para cada registro presente no arquivo. Por meio de cache distribuído, as informações das partições são transmitidas para as máquinas participantes e os dados são particionados. Cada partição é atribuída a um processador, que executa a tarefa Reduce. Na fase Reduce, cada processador ordena os dados localmente. Após a ordenação local, os dados são enviados para a máquina mestre, na qual são concatenados e formam o conjunto final ordenado.

A Figura 2 apresenta um diagrama exemplificando uma execução do algoritmo implementado no Hadoop. Nesse exemplo, representamos a execução do algoritmo em duas máquinas com dois núcleos cada, totalizando quatro unidades de processamento. Primeiramente foram lidos os arquivos (passo 1) e a amostra foi composta por três valores (passo 2). Em seguida, são formados os pares <chave, valor> pela função Map (passo 3). Os dados são divididos nas quatro partições (passo 4). Formadas as partições, os dados são distribuídos aos núcleos de processamento para execução da função Reduce, que ordena localmente os dados (passo 5). Finalmente o mestre agrupa todos os valores, escrevendo o arquivo final (passo 6).

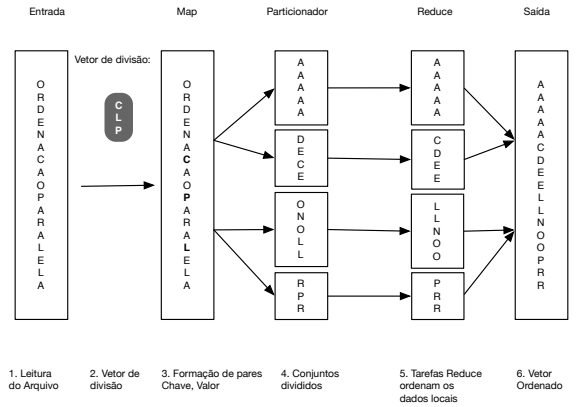


Figura 2. Fluxo de execução do algoritmo Ordenação por Amostragem

IV. Experimentos e Ambiente de Testes

Essa seção apresenta o ambiente de teste, os objetivos dos testes, as métricas de avaliação, bem como as cargas de trabalho utilizadas.

O trabalho foi desenvolvido no Laboratório de Redes e Sistemas (LABORES) do Departamento de Computação (DECOM) do CEFET-MG. Os testes foram realizados em um *cluster* formado por cinco máquinas (número máximo disponível para os testes) com processadores Intel Core 2 Duo de 3.0 GHz, disco rígido SATA de 500 GB 7200 RPM, memória RAM de 4 GB, placa de rede Gigabit Ethernet, sistema operacional Ubuntu 10.04 32 bits, Sun Java JDK 1.6.0 19.0-b09, Apache Hadoop 1.1.0.

Para a realização dos testes foram gerados números aleatórios de acordo com três distribuições: uniforme, normal e Pareto. As distribuições foram geradas por um programa implementado em Java para geração de chaves aleatórias de ponto flutuante. Foram gerados conjuntos contendo entre 10^6 e 10^{10} chaves. Os gráficos da Figura 3 exibem o padrão de frequência das chaves nas três distribuições: uniforme, normal e Pareto, respectivamente.

Os algoritmos Ordenação por Amostragem e Quicksort Paralelo foram implementados, testados e avaliados quanto ao seu desempenho na ordenação e sua escalabilidade. Os algoritmos foram executados em conjuntos de dados de tamanhos variados, os tempos de execução foram medidos e então foram calculadas as métricas *speedup* e eficiência.

O *speedup* (S_p) é uma métrica que mede a escalabilidade de uma aplicação paralela, definida como a razão entre o tempo gasto para executar um algoritmo em uma única máquina e o tempo gasto para executá-lo em p processadores. Em uma situação ideal, o *speedup* é igual a p, indicando que o aumento da capacidade de processamento é diretamente proporcional ao aumento do número de processadores. Contudo, o *speedup* real é diretamente afetado por fatores como comunicação entre processos, granulosidades inadequadas e partes não paralelizáveis de programas [12].

A eficiência (E_p) é a razão entre o *speedup* e o número de processadores [12]. A eficiência avalia quão bem estão sendo utilizados os recursos do sistema. Seu valor ideal é 1 ou 100%, significando que o processamento paralelo alcançou o máximo de eficiência em termos do uso dos recursos computacionais.

A. Experimentos

O conjunto de testes realizado com os algoritmos é descrito a seguir.

1) Estabilidade do Algoritmo: avalia a variabilidade do tempo de resposta na execução do algoritmo, que é diretamente afetada pelo tamanho das partições formadas a partir das decisões dos algoritmos.

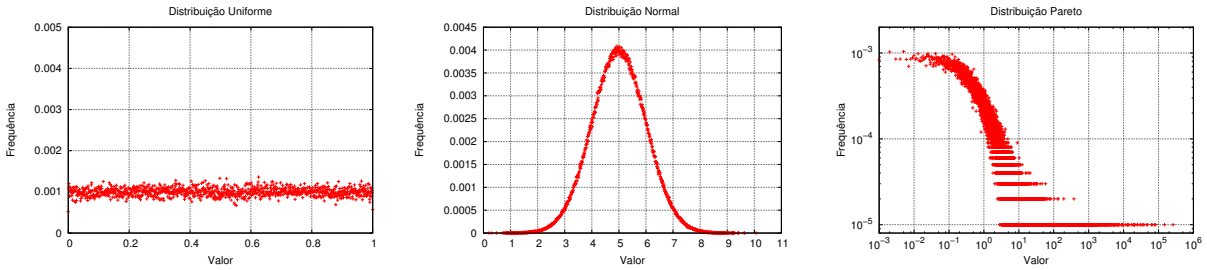
2) Variando o Número de Partições: avalia a influência do número de partições no tempo de ordenação, em cenário com um número constante de máquinas e a mesma quantidade de dados.

3) Variando a Distribuição dos Dados: avalia a influência de distribuições dos dados com características distintas no desempenho dos algoritmos, particularmente na divisão dos dados e na escolha dos valores para os limites das partições.

4) Variando a Quantidade de Dados: avalia o comportamento do algoritmo à medida que o tamanho do conjunto de dados a ser ordenado aumenta, realizando execuções com quantidades crescentes de dados em distribuição uniforme, em cenário com um número fixo de computadores.

5) Variando a Quantidade de Máquinas: avalia a escalabilidade do algoritmo, medida pela diminuição do tempo de ordenação frente ao aumento do número de máquinas. Inclui *speedup* e eficiência.

Para validar o resultado da ordenação realizada pelos algoritmos foi desenvolvido um módulo que verificou



(a) Distribuição Uniforme (b) Distribuição Normal (c) Distribuição Pareto

Figura 3. Distribuições dos dados gerados para ordenação

as chaves escritas após a ordenação completa, comprovando o funcionamento dos algoritmos. Esse programa foi executado após ordenações escolhidas aleatoriamente e confirmou que a ordenação estava correta em todos os casos testados.

V. Resultados

Esta seção apresenta os resultados, de acordo com os testes descritos na seção anterior.

A. *Estabilidade dos Algoritmos*

Os testes de estabilidade medem a variabilidade do tempo de resposta em diversas execuções dos algoritmos para um mesmo conjunto de dados. O tempo de execução é influenciado pelo tamanho das partições formadas, ou seja, pelo balanceamento de carga. Cada algoritmo foi executado dez vezes, utilizando a mesma entrada de dados, contendo 10^8 chaves uniformemente distribuídas. Os testes foram executados em cinco máquinas.

O gráfico da Figura 4 (a) mostra os tempos de cada execução dos dois algoritmos. Os tempos para as dez execuções de cada algoritmo são bastante próximos, o que indica que os algoritmos são estáveis quando executados repetidas vezes com a mesma entrada de dados. As dez execuções do Quicksort Paralelo produzem um tempo médio de 219 segundos, e coeficiente de variação (razão entre o desvio padrão e a média) igual a 0,031. No caso do algoritmo Ordenação por Amostragem, a média dos tempos é 143 segundos e o coeficiente de variação é igual a 0,013. Portanto, oo Quicksort Paralelo apresenta tempos de execução maiores em todos os conjuntos de dados testados, bem como maior variabilidade nos tempos de ordenação.

O gráfico da Figura 4 (b), do tipo boxplot, apresenta o tamanho das partições formadas nos dois algoritmos, em gráficos separados e escalas diferentes para melhor visualização. Observa-se que o Quicksort Paralelo forma partições de tamanhos mais variáveis do que o Ordenação por Amostragem, o que explica a maior variação do tempo de execução.

B. *Variando o Número de Partições*

Nesse teste, o número de divisões feito no processo de ordenação foi alterado, com o objetivo avaliar sua influência no tempo de execução. Foram realizados testes variando o número de partições entre 2, 4, 6 e 8. Cada teste foi executado cinco vezes, em cinco máquinas, com entrada de 10^8 chaves em distribuição uniforme. O gráfico da Figura 5 mostra o tempo médio de ordenação em função do número de partições. Observa-se um aumento significativo no tempo de execução do Quicksort Paralelo à medida que o número de partições aumenta. O contrário ocorre no tempo do Ordenação por Amostragem.

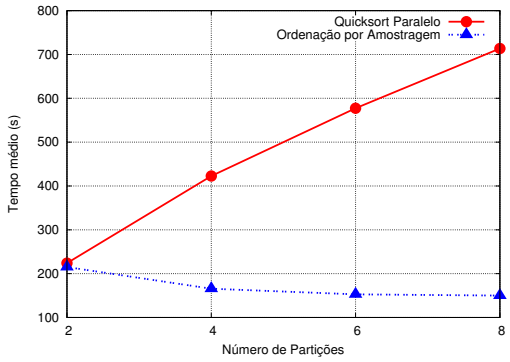
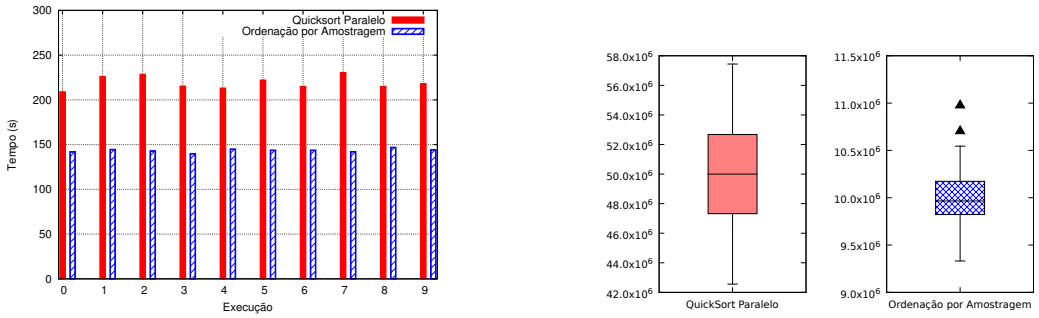


Figura 5. Tempo para ordenação de conjuntos de 10^8 dados em cinco máquinas variando-se o número de partições

O aumento no tempo de ordenação do Quicksort é devido ao maior número de manipulações nos arquivos de dados para realizar as divisões. A Figura 6 apresenta um esquema que mostra esse comportamento. A divisão dos dados de entrada em oito partições resulta em seis manipulações a mais nos arquivos de dados quando comparado a duas divisões, o que explica o maior tempo de execução. Em função desse resultado, nos demais testes o número de partições foi fixado em duas para o Quicksort Paralelo e, para o Ordenação por Amostragem, em um número variável de acordo com o número de núcleos e máquinas.

C. *Variando a Distribuição de Dados*

Os testes variando a distribuição dos dados buscaram avaliar se dados com características distintas influenciam a divisão das partições e alteram o desempenho dos algoritmos. Foram utilizados arquivos de 10^8 chaves geradas



(a) Tempos de execução

(b) Tamanhos das partições

Figura 4. Resultados dos testes de estabilidade para conjuntos de 10^8 chaves em cinco máquinas

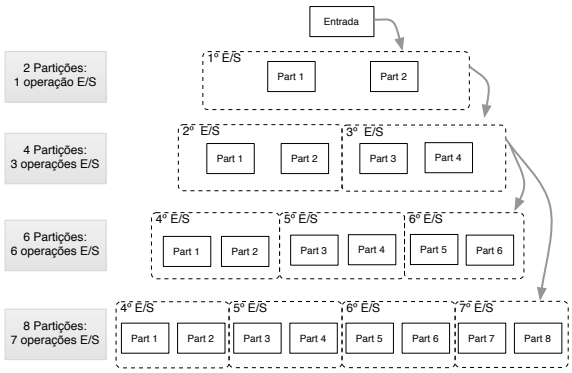


Figura 6. Operações de E/S com o aumento no número de partições

Tabela I
TEMPOS MÉDIOS DE ORDENAÇÃO (SEGUNDOS) PARA CONJUNTOS DE DADOS NAS DISTRIBUIÇÕES UNIFORME, NORMAL E PARETO

Distribuição	Quicksort Paralelo	Ordenação por Amostragem
Uniforme	222	141
Normal	223	143
Pareto	218	143

aleatoriamente de acordo com cada uma das seguintes distribuições: normal, Pareto e uniforme. Para cada distribuição a ordenação foi repetida dez vezes, em cinco máquinas. Os gráficos da Figura 3 apresentam o perfil dos dados.

A Tabela I apresenta o tempo médio de ordenação para cada algoritmo para entradas de dados com as diferentes distribuições. Os resultados mostram que o tempo de ordenação não foi significativamente influenciado pela distribuição dos dados de entrada. Assim, concluímos que os algoritmos são robustos em relação à variabilidade dos dados, uma vez que a distribuição da entrada não teve influência relevante no tempo de execução.

D. Variando a Quantidade de Dados

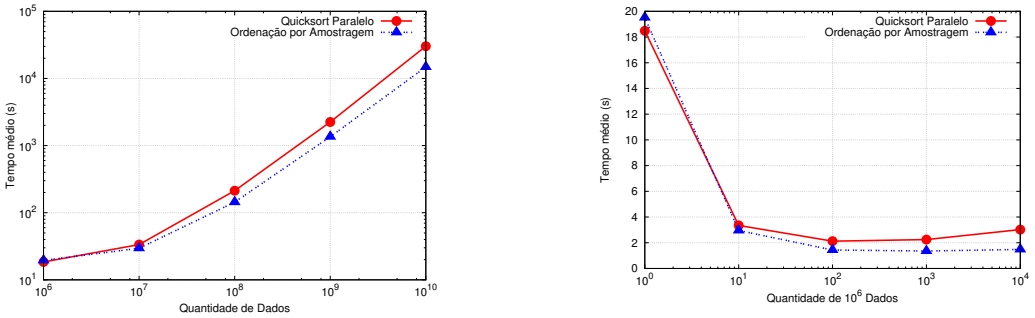
Estes testes buscaram avaliar o comportamento dos dois algoritmos frente ao aumento do tamanho da entrada de dados. Foram feitas execuções com dados uniformemente distribuídos, com quantidades variando de 10^6 a 10^{10}. As ordenações foram feitas em cinco máquinas e o teste foi repetido cinco vezes para cada quantidade de dados. Para o Quicksort Paralelo foram definidas duas partições. Para o algoritmo Ordenação por Amostragem o número de partições foi calculado pela fórmula: $\textit{partições} = \textit{máquinas} \times \textit{núcleos}$. Portanto foram feitas dez partições.

Os tempos médios das ordenações são apresentados no gráfico da Figura 7 (a), com eixos em escala logarítmica para melhor visualização. Os resultados mostram que os tempos do algoritmo Ordenação por Amostragem são entre 12% (para 10^7) e 52% (para 10^{10}) menores do que os tempos do Quicksort. Isso ocorre para todos os conjuntos de dados, exceto para o conjunto contendo 10^6 chaves. Uma possível explicação para esse comportamento é que, para conjuntos pequenos de dados, a sobrecarga de comunicação inserida, pelo algoritmo Ordenação por Amostragem, para gerar a amostra e realizar a divisão da entrada tem influência significativa no tempo total de ordenação desses conjuntos, o que torna o algoritmo mais lento nesses casos.

Os resultados também mostram que a diferença entre os tempos é crescente com o aumento da quantidade de dados, o que evidencia a melhor escalabilidade do algoritmo Ordenação por Amostragem. A maior diferença entre os tempos de ordenação, observada para o conjunto de 10^{10} dados, reflete a distribuição mais equitativa da carga feita pelo algoritmo Ordenação por Amostragem, que resulta em tempos de ordenação menores.

Com a finalidade de dimensionar o *overhead* do algoritmo, isto é, a sobrecarga ou custo adicional de comunicação entre os processadores, assim como sua escalabilidade em relação ao número de dados, foi calculado o tempo médio de ordenação para cada conjunto de 10^6 dados. O gráfico da Figura 7 (b) apresenta o tempo médio de ordenação por megadado (10^6 dados) em função do tamanho da entrada de dados.

Observa-se que com o aumento no tamanho da entrada de dados houve uma melhora significativa no desempenho dos algoritmos, evidenciada pela redução do tempo de ordenação por megadado, para entradas de até 10^9



(a) Ordenação de conjuntos de 10^6 a 10^{10} dados

(b) Tempos para ordenação de conjuntos de 10^6 dados

Figura 7. Tempos médios de ordenação (em segundos) em cinco máquinas

chaves. Para ordenação 10^{10} chaves, no entanto, houve um aumento no tempo relativo de ordenação do Quicksort Paralelo, o que indica uma piora no desempenho para esse caso. É possível que a entrada de dados tenha se tornado demasiadamente grande para o uso de apenas duas partições, o que explica o aumento no tempo.

A diferença no tempo de ordenação de um megadado (10^6 dados) e 10 megadados (10^7) é a mais significativa, com uma redução de 81% no Quicksort Paralelo. Essa diferença pode ser atribuída ao custo intrínseco (*overhead*) do Hadoop, o que indica que ele é mais apropriado para quantidades maiores de dados, isto é, esse custo é amenizado nas execuções maiores.

E. Variando a Quantidade de Máquinas

Os testes variando a quantidade de máquinas foram realizados para avaliar a escalabilidade do algoritmo. Cada teste foi realizado cinco vezes, para conjuntos de 10^8 chaves com distribuição uniforme, que foram ordenadas por grupos de 2, 3, 4 e 5 máquinas. O gráfico da Figura 8 (a) representa a variação dos tempos de ordenação à medida que o número de máquinas aumenta. Podemos observar que o decaimento das curvas é similar para os dois algoritmos, frente ao aumento do número de máquinas. Observa-se ainda que a diminuição do tempo não é linear.

O tempo médio de execução do algoritmo Quicksort Paralelo em cinco máquinas foi cerca de 45% menor que o tempo médio obtido em duas máquinas. Para o Ordenação por Amostragem o percentual reduzido é ainda maior: em cinco máquinas reduz-se 57% o tempo necessário para a execução em duas máquinas. Em todos os casos o coeficiente de variação calculado para os tempos de execução foi menor que 0,031, o que indica tempos de execução bastante homogêneos para os dois algoritmos.

F. Speedup e Eficiência

A fim de avaliar o desempenho do algoritmo foram calculadas as métricas *speedup* S e eficiência E. Neste trabalho não implementamos versões sequenciais dos algoritmos. Portanto, consideramos, como base de cálculo para o *speedup* a execução em quatro processadores (duas máquinas), ao invés da execução sequencial. As fórmulas foram adaptadas para o cálculo das métricas a partir de duas máquinas:

$$S_{real} = \frac{T_{4\ processadores}}{T_{paralelo}} \quad \text{e} \quad E = \frac{S_{real}}{S_{ideal}}$$

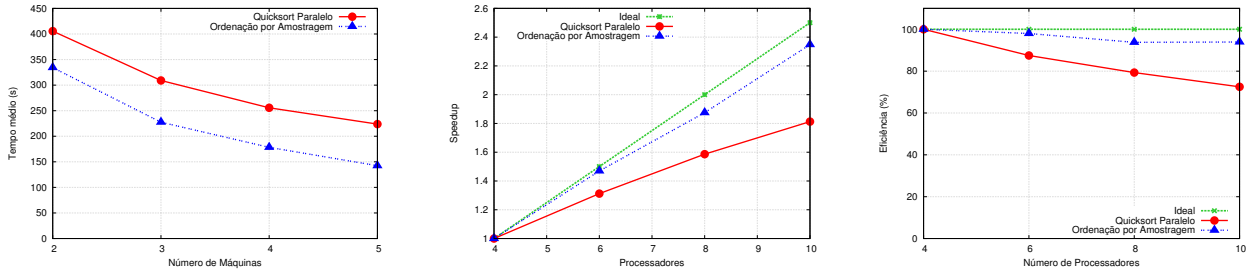
O gráfico da Figura 8 (b) apresenta a curva do *speedup* real dos algoritmos Quicksort Paralelo e Ordenação por Amostragem, além da curva de *speedup* ideal para comparação. Inicialmente todos os valores de *speedup* são iguais. No entanto, com o aumento no número de processadores, o *speedup* real se afasta do *speedup* ideal, fato que ocorre com maior acentuação no algoritmo Quicksort. A variação do *speedup* é um comportamento esperado, uma vez que, em geral, paralelizações introduzem sobrecargas para realizar o balanceamento e a comunicação entre os processos, o que impede que o ganho real seja exatamente proporcional ao aumento no número de processadores.

O gráfico da Figura 8 (c) apresenta a eficiência dos algoritmos paralelos. Observa-se que, à medida que o número de processadores aumenta, o valor da eficiência decresce, mostrando que a eficiência no uso dos recursos é menor para maior número de processadores. Esse fato ocorre mais acentuadamente no algoritmo Quicksort, o que indica que o algoritmo Ordenação por Amostragem apresenta melhor utilização dos recursos, resultante de sua distribuição mais uniforme da computação entre os computadores.

VI. Conclusão

Nesse trabalho apresentamos um estudo comparativo de dois algoritmos de ordenação paralela, a saber, Ordenação por Amostragem e Quicksort Paralelo, ambos implementados em Java, de acordo com o modelo MapReduce, e executados no ambiente Hadoop.

O conjunto de experimentos planejado permitiu uma ampla análise do problema. A partir dos resultados obtidos podemos concluir que: *(a)* os algoritmos são estáveis, pois apresentam tempos de execução muito similares no mesmo cenário; *(b)* os algoritmos formam partições balanceadas, com aproximadamente o mesmo número de elementos, o que permite que o tempo de ordenação seja similar em



(a) Tempos de execução (b) Speedup (c) Eficiência

Figura 8. Resultados para a variação do número de máquinas e núcleos de processamento

diferentes execuções; *(c)* a distribuição dos dados não influencia o tempo de ordenação e o tamanho das partições; *(d)* com o aumento do tamanho do conjunto de dados, os resultados apresentam uma melhoria significativa no tempo de ordenação por megadado (10^6 dados), com uma exceção; *(e)* com o aumento do número de máquinas, o tempo de ordenação é reduzido substancialmente; *(f)* os resultados mostram valores de *speedup* e eficiência sublineares, mas próximos do ideal, especialmente para o algoritmo Ordenação por Amostragem, que teve uma eficiência superior a 93%.

Os resultados apontaram de maneira conclusiva o melhor desempenho e escalabilidade do algoritmo Ordenação por Amostragem, que apresentou os menores tempos de ordenação e uma boa distribuição de trabalho entre as máquinas presentes no *cluster*. O algoritmo Quicksort Paralelo apresentou bons resultados quando utilizado para ordenar quantidades menores de dados, uma vez que não realiza processamento extra, amostrando a entrada, para definir as partições. No entanto, para maiores quantidades de dados, esse algoritmo apresenta desempenho inferior, devido ao número reduzido de partições e uma divisão de trabalho mais limitada entre os processadores. Isso ocorre porque, a cada particionamento dos dados, o algoritmo realiza maior quantidade de manipulações nos arquivos.

Os trabalhos futuros incluem a realização de experimentos com mais máquinas e quantidades maiores de dados, bem como novos estudos no ambiente Hadoop, com implementação de outros algoritmos de ordenação paralela e também a investigação do efeito da configuração do ambiente nos tempos de ordenação.

AGRADECIMENTOS

Os autores agradecem ao INCT InWeb, ao CNPQ e à FAPEMIG pelo apoio.

REFERÊNCIAS

[1] J. Gantz, "The 2011 Digital Universe Study: Extracting Value from Chaos," 2011. [Online]. Available: http://www.emc.com/collateral/demos/microsites/emc-digital-universe-2011/index.htm

[2] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick, "A View of the Parallel Computing Landscape," *Commun. ACM*, vol. 52, no. 10, pp. 56–67, Oct. 2009.

[3] J. Lin and C. Dyer, *Data-Intensive Text Processing with MapReduce*. Morgan & Claypool Publishers, 2010.

[4] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[5] T. White, *Hadoop: The Definitive Guide*, 1st ed. Sebastopol, CA, USA: O'Reilly, June 2009.

[6] Hadoop, " Welcome to Apache Hadoop!" Website, 2012. [Online]. Available: http://hadoop.apache.org

[7] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating MapReduce for Multi-core and Multiprocessor Systems," in *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 13–24.

[8] V. Kale and E. Solomonik, "Parallel Sorting Pattern," in *Proceedings of the 2010 Workshop on Parallel Programming Patterns*, ser. ParaPLoP '10. New York, NY, USA: ACM, 2010, pp. 10:1–10:12.

[9] N. M. Amato, R. Iyer, S. Sundaresan, and Y. Wu, "A Comparison of Parallel Sorting Algorithms on Different Architectures," Texas A & M University, College Station, TX, USA, Tech. Rep., 1998.

[10] L. Cherkasova, "Performance Modeling in MapReduce Environments: Challenges and Opportunities," in *Proceedings of the Second Joint WOSP/SIPEW International Conference on Performance Engineering*, ser. ICPE '11. New York, NY, USA: ACM, 2011, pp. 5–6.

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. Cambridge, MA: MIT Press, 2009.

[12] T. Rauber and G. Rünger, *Parallel Programming for Multicore and Cluster Systems*. Berlin, Heidelberg: Springer Verlag, 2010.