



CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS  
CURSO DE ENGENHARIA DE COMPUTAÇÃO  
Trabalho de Conclusão de Curso

Paula de Moraes Pinhão

## **Ordenação Paralela no Ambiente Hadoop**

Orientadora: Prof<sup>a</sup>. Dr<sup>a</sup>. Cristina Duarte Murta

Belo Horizonte  
2011

## Resumo

Estima-se que a ordenação seja responsável por grande parte do processamento executado em computadores. Muitos programas, tais como compiladores, sistemas operacionais e muitas aplicações usam extensivamente a ordenação para lidar com tabelas e listas. Em consequência, a ordenação é parte vital da computação e um de seus problemas mais estudados. Com a possibilidade do processamento paralelo, versões paralelas dos algoritmos sequenciais e novos algoritmos paralelos de ordenação têm sido desenvolvidos, com o intuito de diminuir consideravelmente o tempo de execução da ordenação e, principalmente, possibilitar a ordenação de uma quantidade maior de dados em um tempo compatível com a capacidade de processamento. Este trabalho apresenta um panorama da ordenação paralela, uma implementação e resultados de desempenho de um algoritmo de ordenação paralela. O modelo de programação paralela utilizado foi o MapReduce. A implementação foi feita utilizando o software aberto Hadoop. Os resultados mostram a escalabilidade do programa em relação à quantidade de dados ordenados e de máquinas utilizadas.

**Palavras-chave:** Ordenação, Programação Paralela, MapReduce, Hadoop, Escalabilidade.

## Abstract

*Sorting may be responsible for much of the processing performed on computers. Many programs, such as compilers, operating systems and many applications use sorting extensively to handle with tables and lists. Consequently, sorting is a vital part of computing and one of its most studied problems. The possibility of parallel processing, including parallel versions of sequential algorithms and new parallel sorting algorithms have been developed with the aim of reducing the execution time of sorting and mainly allow the sorting of a larger amount of data at a time compatible with the processing capacity. This work presents an overview of parallel sorting, an implementation and performance results of a parallel sorting algorithm. The parallel programming model used was MapReduce. The implementation was done using the open source Hadoop. The results show the scalability of the program relating to the amount of data sorted and machines used.*

**Keywords:** *Sorting, Parallel Programming, MapReduce, Hadoop, Scalability.*

# Sumário

<b>Lista de Figuras</b>	<b>3</b>
<b>Lista de Tabelas</b>	<b>6</b>
<b>1 Introdução</b>	<b>7</b>
1.1 Apresentação do Problema . . . . .	7
1.2 Motivação . . . . .	8
1.3 Objetivos . . . . .	10
1.4 Estado da Arte . . . . .	10
1.5 Organização do Texto . . . . .	11
<b>2 Arquitetura Paralela</b>	<b>12</b>
2.1 Classificação de Arquiteturas Paralelas . . . . .	14
2.2 Arquitetura de Processadores <i>Multicore</i> . . . . .	16
<b>3 Programação Paralela</b>	<b>19</b>
3.1 Paralelismo x Concorrência . . . . .	21
3.2 Lei de Amdahl . . . . .	21
3.3 Modelos de Programação Paralela . . . . .	22
3.3.1 Memória Compartilhada . . . . .	23
3.3.2 Memória Distribuída . . . . .	23
3.3.3 Paralelismo de Dados . . . . .	24
3.3.4 <i>Thread e Multithreads</i> . . . . .	24
3.3.5 MapReduce . . . . .	25
3.4 Paralelização de Programas . . . . .	28
<b>4 Ordenação Paralela</b>	<b>30</b>
4.1 Limites da Ordenação Paralela . . . . .	32
4.2 Descrição de Algoritmos de Ordenação . . . . .	33
4.2.1 <i>BubbleSort</i> . . . . .	33
4.2.2 <i>QuickSort</i> . . . . .	34
4.2.3 <i>RadixSort</i> . . . . .	35
4.3 Ordenação Externa . . . . .	37

4.4	Ordenação na Plataforma Hadoop . . . . .	38
4.5	Ordenação por Amostragem . . . . .	38
<b>5</b>	<b>Metodologia e Planejamento dos Experimentos</b>	<b>42</b>
5.1	Ambiente de Desenvolvimento e Testes . . . . .	43
5.2	Planejamento de Testes . . . . .	43
5.2.1	Testes Utilizando <i>Benchmarks</i> de Ordenação . . . . .	43
5.2.2	Testes de Verificação . . . . .	44
5.2.3	Testes de Estabilidade do Sistema . . . . .	44
5.2.4	Testes Variando o Conjunto de Dados . . . . .	44
5.2.5	Testes Variando a Quantidade de Dados . . . . .	44
5.2.6	Testes Variando a Quantidade de Máquinas . . . . .	45
5.3	Geração da Carga de Trabalho . . . . .	45
5.3.1	<i>Benchmarks</i> de Ordenação . . . . .	45
5.3.2	Algoritmo de Ordenação por Amostragem . . . . .	45
5.4	Parâmetros de Balanceamento . . . . .	46
<b>6</b>	<b>Resultados</b>	<b>47</b>
6.1	Testes Utilizando <i>Benchmarks</i> de Ordenação . . . . .	47
6.2	Testes de Verificação do Algoritmo . . . . .	48
6.3	Testes de Estabilidade do Sistema . . . . .	48
6.4	Testes Variando o Conjunto de Dados . . . . .	50
6.5	Testes Variando a Quantidade de Dados . . . . .	51
6.6	Testes Variando a Quantidade de Máquinas . . . . .	54
6.7	Considerações Finais . . . . .	56
<b>7</b>	<b>Conclusões e Perspectivas</b>	<b>58</b>
7.1	Conclusões . . . . .	58
7.2	Trabalhos Futuros . . . . .	61

## List of Figures

2.1	Primeiros computadores paralelos [Robat 2007] . . . . .	12
2.2	Arquitetura SISD . . . . .	15
2.3	Arquitetura SIMD . . . . .	15
2.4	Arquitetura MISD . . . . .	16
2.5	Arquitetura MIMD . . . . .	16
2.6	Arquitetura de processadores multicore [Kogge 2005, Breshears 2009] . . . . .	17

3.1	Modelo de Programação por Memória Compartilhada . . . . .	23
3.2	Modelo de Programação por Memória Distribuída . . . . .	23
3.3	Modelo de Programação por Dados . . . . .	24
3.4	Modelo de Programação por Threads [Silberschatz e Galvin 2004] . . . . .	25
3.5	Contagem de ocorrências de palavras em documentos com o MapReduce . . . . .	26
3.6	Execução do MapReduce [Dean e Ghemawat 2008] . . . . .	27
3.7	Passos para Paralelização de programas [Rauber e Rünger 2010] . . . . .	28
4.1	Comparações realizadas no <i>BubbleSort</i> [Breshears 2009] . . . . .	33
4.2	Bubblesort com <i>pipelining</i> . . . . .	34
4.3	Partições do <i>QuickSort</i> [Breshears 2009] . . . . .	35
4.4	Duas fases do <i>Radix Exchange Sort</i> [Breshears 2009] . . . . .	36
4.5	<i>Straight Radixsort</i> usando dígitos decimais [Breshears 2009] . . . . .	36
4.6	Funcionamento do algoritmo de Ordenação por Amostragem . . . . .	41
5.1	Formato do arquivo de dados gerado pelo <i>TeraGen</i> . . . . .	45
5.2	Esquema utilizado para geração de dados para ordenação pelo Algoritmo de Ordenação por Amostragem . . . . .	46
6.1	Gráfico dos tempos obtidos para 10 execuções de um arquivo com $10^6$ inteiros aleatórios em 4 máquinas . . . . .	48
6.2	Gráfico da relação entre balanceamento e tempo de ordenação para 10 execuções de um arquivo com $10^6$ inteiros aleatórios em 4 máquinas . . . . .	50
6.3	Gráfico dos tempos médios obtidos para 10 execuções de 10 conjuntos de $10^6$ inteiros aleatórios em 4 máquinas . . . . .	50
6.4	Gráfico dos tempos médios obtidos para $10^6$ a $10^{10}$ inteiros aleatórios em 4 máquinas . . . . .	51
6.5	Gráfico dos tempos médios obtidos para cada $10^6$ dados aleatórios em 4 máquinas . . . . .	53
6.6	Gráfico de interpolação para o gráfico da Figura 6.4 . . . . .	53
6.7	Gráfico dos tempos médios obtidos para três execuções de $10^8$ inteiros aleatórios em 2 a 5 máquinas . . . . .	54
6.8	Gráfico da relação entre o número de processadores e o <i>speedup</i> para $10^8$ inteiros aleatórios . . . . .	55
6.9	Gráfico da relação entre o número de processadores e a eficiência para $10^8$ inteiros aleatórios . . . . .	56

## Listas de Tabelas

5.1	Parâmetros de balanceamento utilizados nos testes . . . . .	46
6.1	Resultados do <i>TeraSort</i> para 2 máquinas . . . . .	47
6.2	Resultados do <i>Sort</i> para 4 máquinas . . . . .	47
6.3	Comparação dos resultados do <i>TeraSort</i> e <i>Sort</i> . . . . .	48
6.4	Tempos estatísticos, em segundos, obtidos para 10 execuções de um arquivo com $10^6$ inteiros aleatórios em 4 máquinas . . . . .	49
6.5	Número de elementos estatísticos das partições obtidas para 10 execuções de um arquivo com $10^6$ inteiros aleatórios em 4 máquinas . . . . .	49
6.6	Tempos estatísticos, em segundos, obtidos para 10 execuções de 10 conjuntos de $10^6$ inteiros aleatórios em 4 máquinas . . . . .	51
6.7	Número de elementos estatísticos das partições obtidas para 10 execuções de 10 conjuntos de $10^6$ inteiros aleatórios em 4 máquinas . . . . .	51
6.8	Tempos estatísticos, em segundos, obtidos para $10^6$ a $10^{10}$ inteiros aleatórios em 4 máquinas . . . . .	52
6.9	Tempos estatísticos, em segundos, para $10^8$ inteiros aleatórios em diferentes quantidades de máquinas . . . . .	54
6.10	<i>Speedup</i> para $10^8$ inteiros aleatórios em 4 a 10 processadores . . . . .	55
6.11	Eficiência para $10^8$ inteiros aleatórios em 4 a 10 processadores . . . . .	56

# Capítulo 1

## Introdução

### 1.1 Apresentação do Problema

A computação paralela ou simplesmente parallelismo, em uma definição bastante genérica, consiste no uso de várias unidades de processamento para executar uma tarefa de forma mais rápida [Tanembaum 2007]. Em outras palavras, ela é uma forma de computação em que vários cálculos são realizados simultaneamente, sob o princípio de que grandes problemas geralmente podem ser divididos em problemas menores e, então, resolvidos em paralelo por vários elementos de processamento. Esses devem cooperar entre si, em busca do melhor desempenho por meio da quebra do paradigma de execução sequencial do fluxo de instruções [Almasi e Gottlieb 1989].

Assim, a ordenação paralela consiste no processo de uso de múltiplas unidades de processamento para ordenar coletivamente uma sequência desordenada. A sequência inicial é decomposta em subsequências disjuntas e cada uma é associada a uma única unidade de processamento que realiza a ordenação da subsequência.

Um grande número de aplicações paralelas contém uma fase de computação intensiva na qual uma grande lista de elementos deve ser ordenada com base em algum atributo comum aos elementos. A questão fundamental na implementação de algoritmos de ordenação paralela é como ordenar uma sequência de elementos em múltiplos processadores, de forma a minimizar a redistribuição de chaves e, ao mesmo tempo, permitir que as unidades de processamento realizem a ordenação de modo independente [Kale e Solomonik 2010].

O MapReduce [Ranger et al. 2007, Dean e Ghemawat 2008], criado pela Google em 2003, é um modelo de programação paralela para processamento largamente distribuído de grandes volumes de dados. Seu objetivo é facilitar a programação de aplicativos distribuídos com este perfil. Uma das implementações mais conhecidas e utilizadas do modelo MapReduce é o Hadoop [Hadoop 2010, Venner 2009, White 2009], plataforma de software livre fortemente baseada no *framework* desenvolvido pela Google, criada por Doug Cutting em 2005, e apoiada pela Yahoo!.

A escalabilidade é um assunto extremamente importante em sistemas eletrônicos, bancos de dados, roteadores e redes de computadores. É uma característica que implica desempenho, pois indica a habilidade de um sistema manipular uma porção crescente de

trabalho de forma uniforme, ou estar preparado para crescer. Dessa forma, um sistema é descrito como escalável se permanece eficiente quando há um aumento significativo no número de recursos e na carga de trabalho [Bondi 2000].

O problema investigado neste trabalho consiste na análise da escalabilidade, em termos do número de dados a serem ordenados e de máquinas utilizadas, do algoritmo de Ordenação por Amostragem, descrito na seção 4.5, implementado em MapReduce no ambiente Hadoop.

## 1.2 Motivação

Estima-se que a ordenação seja responsável por mais de oitenta porcento (80%) de todo ciclo de processamento. Muitos programas, tais como compiladores e sistemas operacionais usam extensivamente a ordenação para lidar com tabelas e listas [Rajasekaran e Reif 1989]. Em consequência, a ordenação é parte vital da computação e um dos problemas mais estudados na ciência da computação [Breshears 2009].

Os melhores algoritmos de ordenação sequenciais, tais como o *QuickSort* e o *HeapSort* [Aho e Hopcroft 1974], normalmente têm custo assintótico  $O(n \times \log n)$  para ordenar uma sequência de  $n$  chaves. Esse tempo aumenta acima do linear com o aumento do número de elementos. Quando a informação não cabe na memória principal do computador, a ordenação deve ser efetuada por meio de métodos de ordenação em memória externa cujo custo comum é  $O(n/b \times \log(n/k))$  onde  $b$  é o tamanho do bloco de leitura ou gravação do sistema operacional e  $k$  é o tamanho do *buffer*.

O rápido avanço de aplicações de software que trabalham com um grande volume de dados e necessitam de intenso processamento tem motivado a evolução dos computadores. A Google, por exemplo, processava 20 PetaBytes (PB) por dia em 2008. O Facebook, em 2009, tinha 2,5 PB de dados e processava 15 TeraBytes (TB) por dia. O eBay tinha 6,5 PB de dados e processava 50 TB de dados por dia em 2009. Com a rápida difusão da informação, esses números aumentam a cada dia.

Do ponto de vista do hardware, a indústria observou que o aumento excessivo da frequência de processamento, o *clock rate*, produzia muitos problemas, como superaquecimento dos *chips*, travamento e aumento no custo [Adve et al. 2008]. Em consequência, a computação paralela se tornou recentemente o paradigma dominante nas arquiteturas de computadores, sob a forma de processadores de múltiplos núcleos (*multicore*) [Asanovic et al. 2006, Asanovic et al. 2008].

As principais linhas de produtos dos dois maiores fabricantes de CPU para PC, *Intel Corporation* (Intel) e *Advanced Micro Devices* (AMD), consistem, atualmente, em processadores de múltiplos núcleos. Por isso, é cada vez mais difícil comprar um computador que não seja no mínimo *Dual Core*. Neste tipo de CPU, cada núcleo (*core*) é um núcleo de processamento físico separado, embora esteja integrado a outros núcleos no mesmo *chip*. Desse modo, um programa escrito apropriadamente, ou seja, de forma a expressar o paralelismo, pode dividir uma tarefa em partes e solucioná-las paralelamente, utilizando mais

de um núcleo do processador. O resultado esperado é a diminuição do tempo necessário para a execução do programa.

É senso comum atualmente que a evolução dos computadores, até mesmo dos mais simples e baratos, passará por um caminho onde cada vez mais será explorado o paralelismo [Pasin e Kreutz 2003]. Os processadores Intel com dois e quatro núcleos já são bastante difundidos hoje. Nos próximos anos, a tendência é que o número de núcleos em um *chip* continue a crescer, marcando o lançamento de uma era de computadores muito mais poderosos [Held 2006].

Porém, de nada adianta esse poder computacional gigantesco, inimaginável há poucos anos, se a programação paralela, que permite utilizar os recursos de hardware das máquinas *multicore* de forma eficiente, continuar sendo uma habilidade limitada a pesquisadores que lidam diretamente com supercomputadores. Segundo os fabricantes de processadores, as máquinas *multicore* são o futuro e, por isso, é fundamental aprender a desenvolver programas adequados para elas, isto é, paralelos. Essa é única forma de aproveitar toda a eficiência permitida por tais máquinas [Tally 2007, Sutter e Larus 2005].

Além disso, se não forem criados programas que utilizem os recursos da computação paralela, o desperdício de energia continuará. Isso acontece porque os transistores consomem energia mesmo quando não estão fazendo nada. Programas sequenciais, executando em processadores de múltiplos núcleos, deixam transistores ociosos e, assim, desperdiçam energia e capacidade de processamento. Ou seja, a capacidade de processamento cresceu, mas não é utilizada [Tally 2007].

Deste modo, com o advento do processamento paralelo, versões paralelas dos algoritmos sequenciais e novos algoritmos de ordenação paralela são cada vez mais desenvolvidos, com o intuito de diminuir consideravelmente o tempo de execução dos algoritmos sequenciais de ordenação e de possibilitar a ordenação de uma quantidade maior de dados em um tempo acessível [Rajasekaran e Reif 1989].

A ordenação tem importância adicional para os desenvolvedores de algoritmos paralelos, pois é frequentemente utilizada para realizar permutações de dados em computadores de memória distribuída. Estas operações podem ser usadas para resolver problemas de teoria dos grafos, computação geométrica e processamento de imagens em tempo próximo ao ótimo [Quinn 1994].

Devido ao grande número de algoritmos de ordenação paralela e a variedade de arquiteturas paralelas, estudos experimentais assumem uma importância crescente na avaliação e seleção de algoritmos apropriados para multiprocessadores. Em geral, quanto mais aplicações paralelas vão sendo desenvolvidas, mais inovadores devem ser os algoritmos de ordenação paralela concebidos para suportá-las. A compreensão do contexto e das necessidades da aplicação final é importante na concepção e implementação de algoritmos de ordenação paralela [Kale e Solomonik 2010].

O Hadoop, uma das implementações *open source* mais conhecidas do MapReduce,

é utilizado por grandes empresas, tais como o Facebook, Yahoo!, Twitter, Microsoft e IBM, que armazenam dados da ordem de PetaBytes ( $10^{15}$  Bytes), e por laboratórios de universidades como a *University of Maryland*, *Cornell University*, *University of Edinburg* e Unicamp. Uma lista de usuários do Hadoop pode ser encontrada em [Hadoop 2010]. O Hadoop é usado para controlar o sistema de busca da Yahoo! e definir os anúncios exibidos ao lado dos resultados. Além disso, determina o que as pessoas visualizam na *homepage* do Yahoo!, ajuda a descobrir a distância de conexão ou grau de separação entre usuários no Facebook e administra quarenta bilhões de fotos armazenadas em seus servidores.

A comunidade acadêmica também já vem notando essa rápida popularização da computação paralela e, como prova disso, a *Association for Computing Machinery* (ACM) lançou, no final de 2008, uma nova versão do seu currículo [Cassel et al. 2008], sugerindo, para os cursos de computação, o aumento da relevância da programação concorrente, devido em grande parte ao desenvolvimento de processadores *multicore*. Nesse mesmo documento é previsto ainda que o papel do paradigma concorrente crescerá cada vez mais, o que influenciará outras áreas da computação.

### 1.3 Objetivos

Este trabalho tem como objetivos:

- (i) Estudar a programação paralela, seus algoritmos e suas possibilidades de implementação em ambiente paralelo *multicore*;
- (ii) Implementar e avaliar o desempenho de um algoritmo de ordenação paralela;
- (iii) Estudar e implementar a solução no modelo MapReduce, com o software Hadoop;
- (iv) Investigar a escalabilidade da ordenação paralela no ambiente Hadoop em relação à quantidade de dados a serem ordenados e de máquinas utilizadas.

### 1.4 Estado da Arte

Os algoritmos paralelos para ordenação têm sido estudados desde o começo da Computação Paralela. Um dos primeiros métodos propostos foi o *Bitonic Sort Network*, em 1968 [Batcher 1968]. Desde então, muitos algoritmos de ordenação paralela foram propostos para arquiteturas vetoriais, multiprocessadores e multicomputadores [Hennessy e Patterson 2007, Blelloch 1990, Leighton 1992]. Na década de 80, foi desenvolvido o algoritmo *FlashSort* [Reif e Valiant 1987] que utiliza uma sofisticada técnica de amostragem ao acaso para formar um conjunto de divisão. Uma versão similar a ele é o *SampleSort* [Huang e Chow 1983] que distribui o divisor definido para cada processador. Outros notáveis algoritmos de ordenação são as versões paralelas do *RadixSort*, *QuickSort* [Blelloch 1990, Leighton 1992, Breshears 2009] e *MergeSort* [Cole 1988].

Amato et al. [Amato et al. 1998] apresenta uma avaliação comparativa de desempenho de três algoritmos de ordenação paralela: *BitonicSort*, *SampleSort*, e *RadixSort* paralelo em

três diferentes arquiteturas: uma máquina vetorial com 2.048 elementos de processamento, um hipercubo com passagem de mensagens e 32 processadores e uma máquina de memória distribuída com dez processadores. Ele concluiu que, para cada máquina, a escolha do algoritmo depende do número de elementos a ser ordenado. Para a máquina vetorial, por exemplo, os resultados mostraram que o algoritmo *BitonicSort* foi o mais rápido para tamanhos menores de entrada e que o *RadixSort* paralelo foi o mais rápido para tamanhos maiores de entrada. Já para a arquitetura hipercubo, o *SampleSort* apresentou os melhores resultados para diferentes tamanhos de entrada, uma vez que minimizou o movimento de dados e a comunicação entre processadores, que são consideradas operações caras. Para a máquina de memória distribuída, o *SampleSort* foi o algoritmo mais rápido para menores valores de  $n$  e o *RadixSort* paralelo foi o mais rápido para tamanhos maiores do problema.

Um trabalho de destaque na literatura é o de Leighton [Leighton 1985] que aborda os limites de complexidade e propõe o algoritmo de ordenação paralela *ColumnSort*. Outro importante trabalho é o de Shi e Schaeffer [Shi e Schaeffer 1992] que apresenta um novo algoritmo de ordenação paralela adequado para multiprocessadores denominado PSRS (*Parallel Sorting by Regular Sampling*) que é baseado em amostragem regular dos dados para garantir a seleção de um bom pivô. Já Kale e Solomonik [Kale e Solomonik 2010] discutem os padrões de ordenação paralela, as questões envolvidas na sua implementação e versões paralelas de algoritmos como o *QuickSort* e *RadixSort*.

Jim Gray [Gray 1998] criou o *Sort*, um dos mais conhecidos *benchmarks* de ordenação de dados para Hadoop, que consiste em um conjunto de vários *benchmarks* relacionados, cada um com suas regras. Por exemplo, a categoria *MinuteSort* deve ordenar a maior quantidade possível de dados em um minuto e a *GraySort* deve ordenar mais que cem TeraBytes em pelo menos uma hora [White 2009].

Outra aplicação de destaque para ordenação de dados com Hadoop é o *TeraSort*, criado por Owen O' Malley [O'Malley e Murthy 2009], com o intuito de participar da competição de ordenação chamada *GraySort* [Gray 1998]. Em 2009, o *TeraSort* foi o campeão dessa competição nas categorias *MinuteSort*, ao ordenar 500 GB em 59 segundos em 1.406 nós, e *GraySort*, ao ordenar 100 TB em 173 minutos (2,88 horas) em 3.452 nós. Nesse mesmo ano, o *TeraSort* ordenou um PetaByte em 975 minutos (16,25 horas) em 3.658 nós.

## 1.5 Organização do Texto

A continuidade deste texto está organizada como descrito a seguir. Nos capítulos 2, 3, e 4 são apresentados os principais conceitos de arquitetura paralela, programação paralela e ordenação paralela, respectivamente, necessários à compreensão e desenvolvimento deste trabalho. No capítulo 5 é discutida a metodologia adotada no trabalho e a forma com que ele foi desenvolvido. O capítulo 6 apresenta e discute os resultados obtidos. O capítulo 7 aborda as conclusões obtidas no trabalho, assim como as perspectivas de continuação do mesmo.

## Capítulo 2

# Arquitetura Paralela

Computadores paralelos têm sido usados há pelo menos três décadas e muitas arquiteturas alternativas diferentes têm sido propostas e utilizadas. Em geral, um computador paralelo pode ser caracterizado como uma coleção de elementos de processamento que podem comunicar-se e cooperar para resolver problemas rapidamente [Almasi e Gottlieb 1994, Asanovic et al. 2006].

No passado, os esforços de computação paralela se mostraram promissores e reuniram investimento. Um avanço significativo veio com o supercomputador Cray-1, da *Cray Research*, em 1971, apresentado na Figura 2.1a, que foi a primeira máquina a implementar o conceito de *pipeline*, cujo processador executava uma instrução dividindo-a em partes, como em uma linha de montagem de um carro. Desse modo, enquanto a segunda parte de uma instrução estava sendo processada, a primeira parte de outra instrução começava a ser trabalhada. A evolução seguinte foi a denominada máquina vetorial, cujo processador trabalhava com mais de um conjunto de dados ao mesmo tempo. Um pouco depois apareceram máquinas com múltiplos processadores, como a *Connection Machine*, com 65.536 processadores, representada na Figura 2.1b.

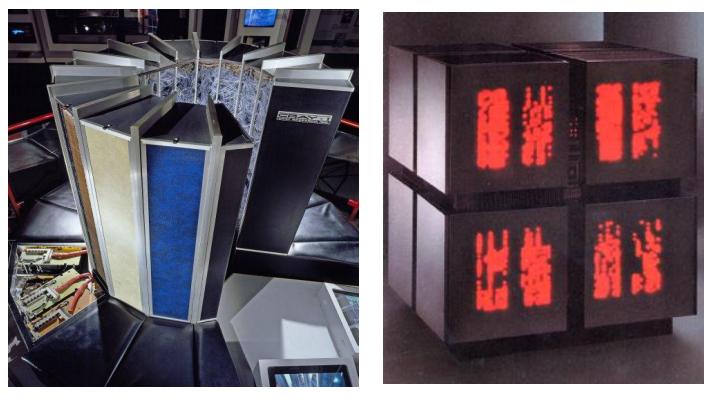


Figura 2.1: Primeiros computadores paralelos [Robat 2007]

Entretanto, a computação em um único processador sempre prevaleceu, pois os compu-

tadores paralelos eram muito caros. O Cray-1, por exemplo, custava mais de cinco milhões de dólares. Esse cenário é bem diferente dos dias de hoje, já que, como argumentado em [Asanovic et al. 2006], a computação está dando um passo irreversível em direção às arquiteturas paralelas.

O número de transistores em um *chip* de computador pode ser usado como uma estimativa de sua complexidade e desempenho [Asanovic et al. 2006]. Em [Moore 1965], Gordon Moore, co-fundador da Intel, observou que o número de transistores de um típico *chip* de processador dobrava a cada 18 ou 24 meses. Por muito tempo o aumento no número de transistores foi acompanhado por um aumento na velocidade de *clock* e, em consequência, uma melhoria no desempenho dos sistemas de computação. Depois de mais de quarenta anos, a lei de Moore continua válida, uma vez que a densidade de semicondutores continua a aumentar.

O aumento no número de transistores pode ser utilizado para melhorias na arquitetura, tais como unidades funcionais adicionais, maior quantidade e tamanho de *caches*, assim como mais registros. Essas melhorias reduzem o tempo médio de execução de uma instrução. Quatro fases de tendências de *design* de microprocessadores podem ser observadas [Culler e Singh 1999], que são conduzidas principalmente pelo uso de paralelismo interno:

1. **Paralelismo no nível de bit (BLP):** consiste no aumento do tamanho da palavra do processador, isto é, da sequência de bits de tamanho fixo processada em conjunto em uma máquina. Até cerca de 1986, o tamanho dos registradores aumentou gradualmente de 4 para 32 bits, o que permitiu operações eficientes em números em um intervalo maior. Na década de 90, surgiram os microprocessadores de 64 bits que vêm gradativamente substituindo os de 32 bits por proverem precisão suficiente para números de ponto flutuante e compreenderem um espaço de endereços suficientemente grande de 264 bytes.
2. **Paralelismo no nível de instrução (ILP):** comprehende técnicas de execução simultânea de instruções de programas no processador. Uma delas é a *pipeline* que divide a execução de cada instrução em N estágios, o que permite que até N instruções estejam em execução ao mesmo tempo, cada uma em seu determinado estágio. Outra técnica muito conhecida é a execução de mais de uma instrução simultaneamente, no mesmo ciclo de *clock*. As instruções são agrupadas apenas se não houver dependência de dados entre elas. É utilizado pelos processadores superescalares.
3. **Paralelismo no nível de dados (DLP):** corresponde a realizar a mesma operação em múltiplos dados simultaneamente. Um exemplo clássico é a execução de uma operação de modificação em uma imagem, como iluminação, na qual cada *pixel* é processado independente dos outros ao redor. Dessa forma, vários *pixels* podem ser modificados simultaneamente com a mesma função de modificação. Essa técnica é

muito utilizada na computação gráfica, em métodos de criptografia, em operações que envolvem matrizes e processamento de vetores. É explorado pelos processadores vetoriais.

4. **Paralelismo no nível de tarefas (TLP):** refere-se ao ato de executar diferentes tarefas no mesmo ou em diferentes conjuntos de dados, simultaneamente. Tais tarefas podem ser definidas por processos ou *threads*. Esse é o nível de paralelismo mais utilizado e é característico dos multiprocessadores ou *multicores*. Ele pode ser frequentemente encontrado em aplicações que precisam executar tarefas independentes, como acessos à memória, simultaneamente. Esses tipos de aplicações são encontradas em máquinas que têm uma alta carga de trabalho, tais como servidores Web.

Por volta de 2003, a velocidade de *clock* atingiu um limite físico: frequências de *clock* acima de 5GHz ocasionavam o derretimento dos *chips*. Os arquitetos foram então forçados a encontrar um novo paradigma para sustentar o crescente aumento de desempenho e reduzir o consumo de energia. A solução encontrada pela indústria foi substituir o único processador ineficiente por muitos processadores eficientes no mesmo *chip*. Dessa forma, iniciou-se a produção de *chips* multiprocessadores com múltiplos independentes núcleos (*cores*) de processamento por *chip*. São os chamados processadores *multicore*, introduzidos no mercado em 2005. Depois disso, a indústria de microprocessadores declarou que seu futuro estava em computação paralela, com uma duplicação do número de núcleos de processamento a cada nova geração de microprocessadores. Tendência essa que continuará no futuro previsível [Leiserson e Mirman 2008, Asanovic et al. 2009].

O uso de múltiplos núcleos em um único processador é considerado como a abordagem mais promissora, pois integra vários núcleos de processamento independentes, com uma arquitetura relativamente simples em um único *chip* de processador. Essa abordagem tem a vantagem adicional de que o consumo de energia de um *chip* processador pode ser reduzido, se necessário, desligando núcleos de processamento não utilizados durante os períodos ociosos [Held 2006].

## 2.1 Classificação de Arquiteturas Paralelas

A possibilidade de executar operações paralelas depende fortemente da arquitetura da plataforma de execução [Rauber e Rünger 2010], que determina como as operações de um programa podem ser mapeadas para os recursos disponíveis e, assim, obter a execução paralela.

Um modelo simples de categorização de todos os computadores foi proposto por Flynn em 1966 e ainda é útil hoje [Flynn 1966]. Ele considerou o paralelismo ao nível de instrução e de dados e, então, distingui quatro categorias de computadores:

1. **SISD (*Single-Instruction, Single-Data*):** um único processador executa um único fluxo de dados, para operar em dados armazenados em uma única memória,

como apresentado na Figura 2.2. Esse modelo corresponde ao convencional computador sequencial, de acordo com o modelo de Von Neumann.

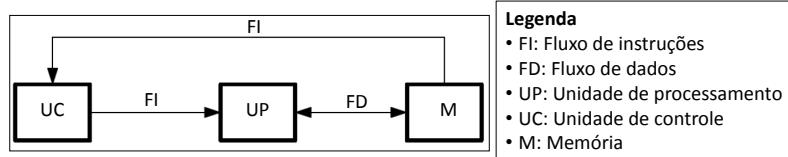


Figura 2.2: Arquitetura SISD

2. **SIMD (Single-Instruction, Multiple-Data):** a mesma instrução é executada por múltiplos processadores usando diferentes fluxos de dados. Computadores dessa categoria exploram o paralelismo no nível de dados aplicando as mesmas operações para múltiplos itens de dados em paralelo. Como representado na Figura 2.3, cada processador tem sua própria memória de dados, mas há uma única instrução de memória e unidade de controle que agrupa e distribui as instruções. Segundo [Rauber e Rünger 2010, Hennessy e Patterson 2007], essa abordagem pode ser muito eficiente para aplicações que apresentam um significativo paralelismo no nível de dados. Exemplos são aplicações multimídia e algoritmos de computação gráfica para gerar visões realistas em três dimensões de ambientes gerados por computador. Os processadores matriciais e vetoriais são os principais tipos de máquinas SIMD.

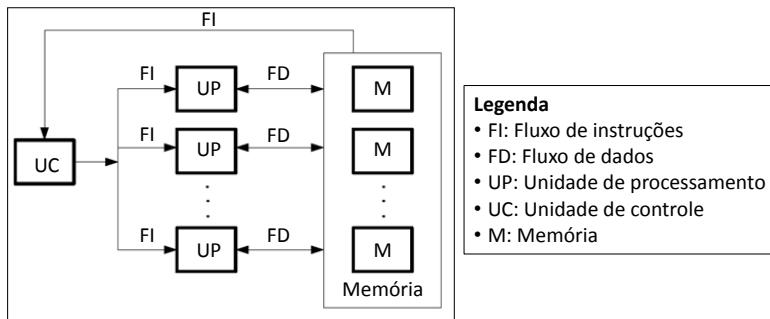


Figura 2.3: Arquitetura SIMD

3. **MISD (Multiple-Instruction, Single-Data):** múltiplas unidades de processamento executam operações diferentes sobre o mesmo conjunto de dados, como mostra a Figura 2.4. De acordo com [Rauber e Rünger 2010, Hennessy e Patterson 2007], essa abordagem é muito restritiva e nenhum computador comercial desse tipo jamais foi produzido.

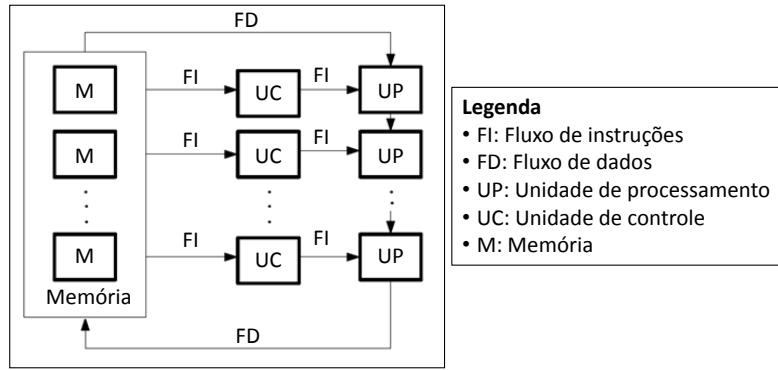


Figura 2.4: Arquitetura MISD

4. **MIMD (Multiple-Instruction, Multiple-Data)**: cada processador busca suas próprias instruções e opera sobre seus próprios dados, de maneira independente, como representado na Figura 2.5. Computadores MIMD exploram o paralelismo no nível de tarefas ou *threads*, uma vez que múltiplas *threads* operam em paralelo. Segundo [Hennessy e Patterson 2007], o paralelismo no nível de tarefas é mais flexível que o de dados e, assim, ele é geralmente mais aplicado. Exemplos desse modelo são os processadores *multicore* e sistemas de *cluster*. A maioria das arquiteturas de computadores paralelos são baseados no conceito MIMD [Rauber e Rünger 2010].

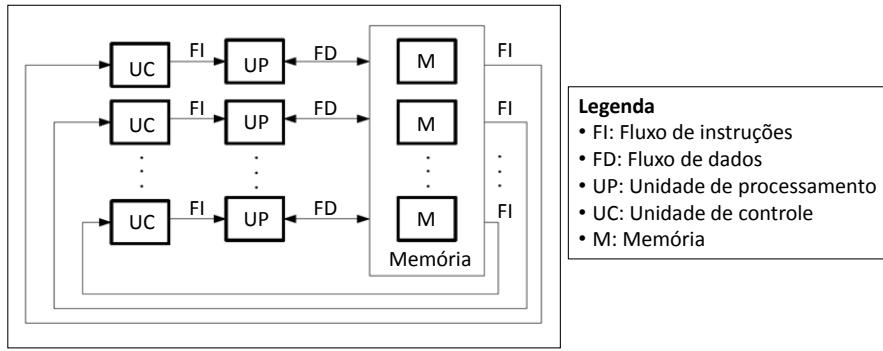


Figura 2.5: Arquitetura MIMD

## 2.2 Arquitetura de Processadores *Multicore*

Há muitas variações de *design* para processadores *multicore*, diferindo no número de *cores*, na estrutura e tamanho de *caches*, no acesso de *cores* aos *caches* e no uso de componentes heterogêneos. A partir de uma visão de alto nível, três diferentes níveis de arquitetura podem ser diferenciados, além das organizações híbridas [Kogge 2005, Breshears 2009]. Esses níveis são apresentados na Figura 2.6.

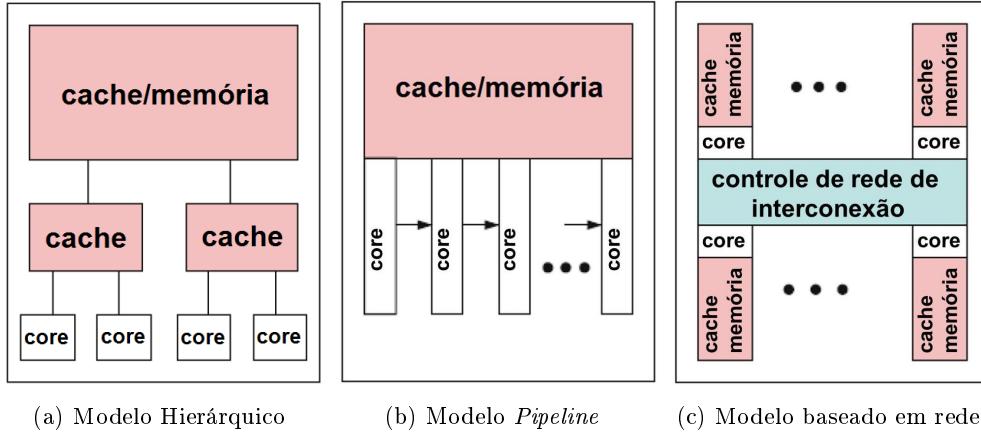


Figura 2.6: Arquitetura de processadores multicore [Kogge 2005, Breshears 2009]

1. **Modelo Hierárquico:** múltiplos núcleos compartilham múltiplos *caches*. Os *caches* são organizados em uma configuração de árvore e o seu tamanho aumenta das folhas para a raiz. A raiz representa a conexão com a memória externa. Assim, cada núcleo pode ter um *cache* L1 separado e compartilha o *cache* L2 com outros núcleos. Todos os núcleos compartilham uma memória externa comum, resultando em uma escala de três níveis. Essa pode ser estendida para mais níveis por meio da utilização de subcomponentes adicionais para conectar os *caches* de um nível com o de outro. Esse modelo pode ser visualizado na Figura 2.6a. Uma área de uso típico desse modelo é o multiprocessamento simétrico (SMP) no qual os processadores compartilham a mesma memória, o que permite mover as tarefas entre eles de modo a tornar a carga de trabalho o mais eficiente possível. Esse modelo também é frequentemente usado em processadores de *desktop* padrão ou servidores. Exemplos são os processadores da Intel Xeon, a arquitetura IBM Power6 e a família AMD Opteron.
2. **Modelo *Pipeline*:** os elementos de dados são processados por vários núcleos de execução de maneira *pipeline*, isto é, cada núcleo realiza etapas específicas de processamento em cada elemento de dados. Os elementos de dados entram no *chip* processador por meio de uma porta de entrada e são passados sucessivamente para núcleos diferentes, até que os elementos de dados processados deixem o último núcleo e o *chip* processador inteiro através de uma porta de saída. Esse modelo está representado na Figura 2.6b. Ele é útil para aplicações nas quais a mesma sequência de operações tem de ser aplicada a uma longa sequência de elementos de dados. Processadores de rede utilizados em roteadores e processadores gráficos executam esse modelo. Dentre eles, estão os processadores Xelerator X10 e X11 fabricados pela *Xelerated Packet Devices AB*.
3. **Modelo baseado em rede:** os núcleos de um *chip* processador, suas *caches* locais e memórias são conectados por uma rede de interconexão com outros núcleos do *chip*. A

transferência de dados entre os núcleos é realizada por meio da rede de interconexão, que também pode fornecer suporte para a sincronização dos núcleos, como mostra a Figura 2.6c. Um exemplo desse modelo é o processador Intel Teraflop, que tem sido projetado pelo programa de pesquisa Intel Tera-scale Computing [Held 2006]. Esse programa aborda os desafios da construção de *chips* de processamento com dezenas a centenas de núcleos de execução, incluindo o projeto do núcleo, gerenciamento de energia, *cache* e hierarquia de memória. O processador Teraflop desenvolvido como um protótipo contém 80 núcleos. Cada núcleo pode executar operações de ponto flutuante e contém um *cache* local, assim como um roteador para executar a transferência de dados entre os núcleos e a memória principal. Há núcleos adicionais para processamento de dados de vídeo, criptografia e computação gráfica.

A arquitetura paralela foi construída há muito tempo, mas somente recentemente podemos experimentá-la a custo razoável, ampliando seu acesso. A literatura indica que estamos experimentando as primeiras versões de arquiteturas *multicore* e a tendência é de que elas evoluam cada vez mais, marcando o início de uma era de computadores cada vez mais poderosos. O estudo da programação paralela torna-se fundamental neste contexto, pois essa é a única forma de aproveitar toda a eficiência permitida por tais máquinas [Pasin e Kreutz 2003, Sutter e Larus 2005, Tally 2007, Leiserson e Mirman 2008, Breshears 2009, Asanovic et al. 2009].

## Capítulo 3

# Programação Paralela

Processadores *multicore* são uma realidade atualmente e tendem a se desenvolver e difundir cada vez mais [Breshears 2009, Leiserson e Mirman 2008, Asanovic et al. 2009, Sutter e Larus 2005]. As principais linhas de produtos dos dois maiores fabricantes de CPU para PC, Intel e AMD, consistem, atualmente, em processadores de múltiplos núcleos. Dessa forma, é cada vez mais difícil comprar um computador que não seja *multicore*.

Segundo [Breshears 2009], para ser um programador diferenciado e empregável é imprescindível aprender a escrever códigos concorrentes, pois essa é a única maneira de aproveitar completamente as vantagens proporcionadas pelos processadores *multicore* e conquistar futuras melhorias de desempenho. Essas melhorias podem ser estimadas potencialmente em execução de aplicações na metade do tempo usando dois núcleos; em um quarto do tempo de execução em quatro núcleos; em um oitavo do tempo de execução em oito núcleos; e assim por diante. Isso soa muito melhor do que a diminuição de 20 a 30% no tempo de execução obtida pelo uso de um processador novo, mais veloz.

Além disso, se não forem criados programas que utilizem os recursos da Computação Paralela, o desperdício de energia continuará. Isso acontece porque os transistores consomem energia mesmo quando não estão fazendo nada. Programas sequenciais, executando em processadores de múltiplos núcleos, deixam transistores ociosos e, assim, desperdiçam energia e capacidade de processamento. Ou seja, a capacidade de processamento cresceu, mas não é utilizada [Tally 2007].

Nas palavras do guru do Microsoft C++, Herb Sutter, para desenvolvedores de software, o “almoço grátis”, que era o aumento anual de desempenho sob a forma de maior velocidade de processamento, acabou. Devido ao fato da programação *multicore* diferir bastante da tradicional programação sequencial, os desenvolvedores de software enfrentam um desafio sem precedentes para os seus investimentos substanciais em bases de código sequenciais legadas. A cada geração de processadores *multicore*, alarga-se a lacuna entre o potencial oferecido pelo hardware e o desempenho que pode ser proporcionado pelas aplicações de hoje. Por esse motivo, os desenvolvedores de software devem identificar e adotar plataformas de software *multicore* que os permitam explorar todas as capacidades de arquiteturas *multicore*, engajar-se em treinamentos massivos para desenvolverem as habilidades necessárias para a programação *multicore* e reformular toda a cadeia de ferramentas de

software, desde a depuração até as versões, para permitir o desenvolvimento de software *multicore* de confiança.

Em contraste com a programação sequencial, há muito mais detalhes e diversidades na programação paralela e uma programação dependente de máquina pode resultar em grande variedade de diferentes programas para o mesmo algoritmo [Rauber e Rünger 2010]. A programação concorrente é difícil [Sutter e Larus 2005]. As linguagens e ferramentas de hoje são inadequadas para transformar aplicações em programas paralelos, e também é difícil encontrar paralelismo em aplicações *mainstream*, ou seja, populares. A programação paralela exige que os programadores pensem de uma maneira difícil e diferente da que estão acostumados. É preciso pensar em múltiplos fluxos executando ao mesmo tempo e em como coordenar todos os fluxos, a fim de completar uma dada computação. Além disso, é preciso preocupar-se com todo um novo conjunto de erros e problemas de desempenho que não têm equivalente na programação serial, tais como disputa de dados e *deadlocks* [Breshears 2009].

Desenvolver software paralelo introduz três desafios [Leiserson e Mirman 2008]:

1. **Conquistar bom desempenho na aplicação:** os desenvolvedores de software precisam de uma estimativa dos impactos das alterações realizadas em seus códigos, incluindo baixo *overhead* em um único núcleo, pois assim poderiam evitar reescrever seus códigos a cada vez que o número de núcleos aumentasse. A comunicação e a sincronização entre diferentes subtarefas é tipicamente uma das maiores barreiras para atingir grande desempenho em programas paralelos. O aumento da velocidade como resultado do paralelismo (*Speedup*) é dado pela lei de Amdahl [Amdahl 1967], apresentada em 1967, descrita na seção 3.2.
2. **Assegurar confiabilidade de software:** quando paralelismo é introduzido em uma aplicação, ela se torna vulnerável às condições de corrida. Essas ocorrem quando tarefas de software concorrente acessam uma localização de memória compartilhada e pelo menos uma delas armazena um valor na localização. Dependendo da ordem de execução das tarefas, o software pode se comportar de forma diferente. Lidar com situações como essas é particularmente desafiante, uma vez que esses *bugs* são não determinísticos e assíncronos, o que os torna difíceis de evitar, encontrar durante testes e, em alguns casos, caros para resolver. A ordem de execução (ou intercalação) de múltiplas *threads* pode ser tão perfeita que os erros não ocorrem, mas se alguma mudança for feita na plataforma de execução que altere a intercalação correta de *threads*, os erros podem começar a aparecer. Mesmo se nenhuma alteração de hardware for feita, execuções consecutivas da mesma aplicação, com as mesmas entradas, podem produzir resultados diferentes sem nenhuma razão aparente. Há ferramentas de software especificamente projetadas para rastrear e identificar erros e questões de desempenho dentro do código da *thread* [Breshears 2009].
3. **Minimizar o tempo de desenvolvimento:** a complexidade de desenvolver soft-

ware *multithread* pode exigir treinamento de pessoal da equipe de desenvolvimento para o aprimoramento de novas habilidades de programação ou redefinição da equipe. Esses fatores podem colocar enorme pressão no cronograma de desenvolvimento e introduzir riscos.

### 3.1 Paralelismo x Concorrência

Os termos “paralelo” e “concorrente” têm sido cada vez mais disseminados desde o lançamento dos processadores *multicore* de propósito geral. Entretanto, mesmo antes disso, já se verificava alguma confusão sobre esses termos em outras áreas da computação [Breshears 2009].

Um sistema é concorrente se é capaz de executar/implementar duas ou mais ações em progresso ao mesmo tempo. Um sistema é paralelo se provê suporte para duas ou mais ações executando simultaneamente. O conceito chave que diferencia essas duas definições é “em progresso”.

A aplicação concorrente terá duas ou mais *threads*, em andamento em algum momento, que serão trocadas nos estados ativada e não ativada pelo sistema operacional em um processador único núcleo. Essas *threads* estarão em progresso, cada uma no meio da sua execução, ao mesmo tempo. Na execução em paralelo, deve haver múltiplos núcleos disponíveis na plataforma de computação. Nesse caso, as duas ou mais *threads* podem ter, cada uma, um núcleo separado e executar simultaneamente.

Pode-se então observar que o paralelismo é um caso específico de programação concorrente, isto é, é possível escrever uma aplicação concorrente que usa múltiplas *threads* ou processos, mas se não houver múltiplos núcleos para execução, o código não poderá ser executado em paralelo. Assim, a programação concorrente e a concorrência englobam toda programação e execução de atividades que envolvem múltiplos fluxos de execução implementados a fim de resolver um único problema.

Portanto, a programação concorrente e o desenvolvimento de algoritmos concorrentes assumem que o código resultante é capaz de executar em um núcleo único ou múltiplos núcleos, sem nenhuma mudança drástica. Pode-se falar sobre a execução paralela de códigos concorrentes, mesmo que o modelo de implementação seja *threads*, desde que se tenha processadores *multicore* disponíveis que permitam executar as múltiplas *threads*. O termo “paralelização” denota o processo de conversão das aplicações seriais em aplicações paralelas.

### 3.2 Lei de Amdahl

A lei de Amdahl demonstra que o ganho de desempenho que pode ser obtido melhorando uma parte do sistema é limitado pela fração de tempo em que essa parte é usada pelo sistema. Ela pode ser utilizada para calcular o ganho de desempenho de uma máquina. Esse ganho de desempenho é também denominado fator de aceleração ou *speedup* e pode ser entendido como a medida de como a máquina se comporta após a implementação de uma

melhoria em relação ao seu comportamento anterior. Quando a melhoria consiste em ter mais processadores, o *speedup* ( $s_p$ ) pode ser definido pela relação:  $S_p = T_{sequencial}/T_{paralelo}$ , isto é, o tempo de execução antes da melhora dividido pelo tempo de execução após a melhora, onde  $p$  é o número de processadores [Hennessy e Patterson 2007, Amdahl 1967].

O *speedup*, portanto, fornece um indicador para o aumento da velocidade resultante da utilização de uma máquina paralela. Para um computador paralelo com  $n$  processadores, o *speedup* ideal seria  $n$  (*speedup linear*) [Amdahl 1967].

Outra medida de desempenho para programas paralelos é a eficiência. Ela define quanto os processadores estão sendo empregados para o processamento paralelo. A eficiência é dada por:  $E_p = s_p/p$ , onde  $p$  é o número de processadores. No caso ideal (*speedup = p*), a eficiência seria máxima e teria valor um (100%).

### 3.3 Modelos de Programação Paralela

Um modelo de programação descreve um sistema de computação paralela em termos da semântica da linguagem de programação ou do ambiente de programação. Seu objetivo é fornecer um mecanismo com o qual o programador pode especificar programas paralelos. Ele é influenciado pela arquitetura e linguagem, compilador, ou pelas bibliotecas de execução. Portanto, há diversos modelos de programação paralela, até mesmo para a mesma arquitetura. Há vários critérios pelos quais os modelos de programação paralela podem diferir [Rauber e Rünger 2010]:

- O nível de paralelismo que é explorado na execução paralela, por exemplo, nível de instrução, comando, procedimento ou *loops* paralelos;
- A especificação implícita ou explícita (definida pelo usuário) do paralelismo;
- A forma como as partes do programa paralelo são especificadas;
- O modo de execução das unidades paralelas (SIMD ou SPMD, síncrono ou assíncrono, definidos na seção 2.1);
- Os modos e padrões de comunicação entre as unidades de computação para a troca de informações (comunicação explícita ou variáveis compartilhadas);
- Os mecanismos de sincronização para organizar a computação e comunicação entre as unidades paralelas.

Os modelos tradicionais de programação paralela são: memória compartilhada, passagem de mensagens, paralelismo de dados e de tarefas (*threads/multithreads*). Além desses, existe um novo modelo denominado MapReduce implementado de maneira *open source* pelo Hadoop. Cada um destes modelos é descrito a seguir.

### 3.3.1 Memória Compartilhada

No ambiente de memória compartilha, múltiplos processadores compartilham o espaço de endereçamento de uma única memória. A comunicação entre os processos é implícita, pois a memória é acessível diretamente por todos os processadores.

As mudanças em um endereço de memória feitas por um processador são visíveis pelos demais, o que torna a programação mais simples. Entretanto, é necessária uma sincronização explícita, por meio do uso de barreiras e *locks*, para evitar problemas de inconsistência de dados resultante do acesso mútuo à memória. Esse procedimento pode incluir erros não determinísticos, muitas vezes difíceis de detectar e corrigir. Além disso, multiprocessadores com memória compartilhada normalmente sofrem com o aumento da contenção e a alta latência no acesso à memória, o que muitas vezes resulta em baixa escalabilidade e desempenho.

A implementação desse modelo pode ser feita pelos compiladores nativos do ambiente. Sua representação é apresentada na Figura 3.1.

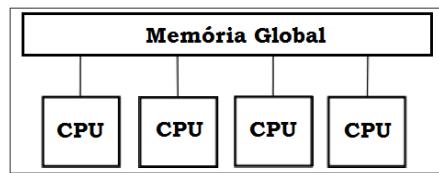


Figura 3.1: Modelo de Programação por Memória Compartilhada

### 3.3.2 Memória Distribuída

O ambiente de memória distribuída consiste em vários processadores, cada um com sua própria memória, interconectados por uma rede de comunicação, como representado na Figura 3.2a.

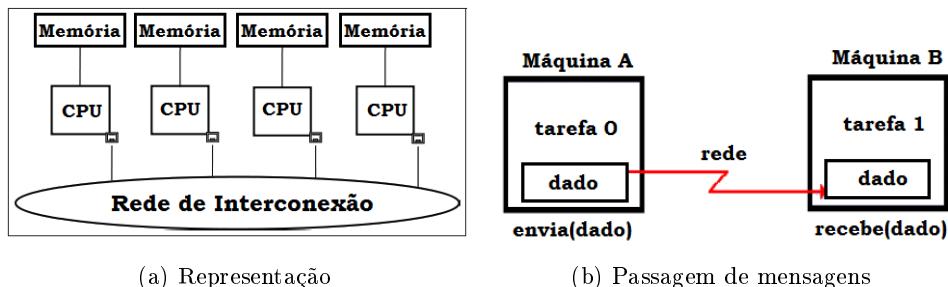


Figura 3.2: Modelo de Programação por Memória Distribuída

As tarefas compartilham dados por meio de comunicação de envio e recebimento de mensagens (“*message-passing*” ou passagem de mensagens). Essa passagem de mensagens, representada na Figura 3.2b, pode ser realizada por meio de bibliotecas como a MPI (*Message Passing Interface*) e a PVM (*Parallel Virtual Machine*).

Nesse modelo, não há limites para o número de processadores e cada processador acessa, sem interferência e rapidamente, sua própria memória. Porém, a programação é complexa, pois o programador é responsável pela comunicação entre os processadores. Além disso, diversos fatores afetam o desempenho da aplicação, tais como o número de vezes que uma mensagem precisa ser copiada e o seu tamanho, a banda de rede, o volume de concorrência, o gerenciamento das passagens de mensagens e a latência do meio de interconexão.

### 3.3.3 Paralelismo de Dados

Modelo de programação no qual as várias tarefas realizam operações em elementos distintos de dados, simultaneamente, e então trocam dados globalmente. Essa representação pode ser visualizada na Figura 3.3.

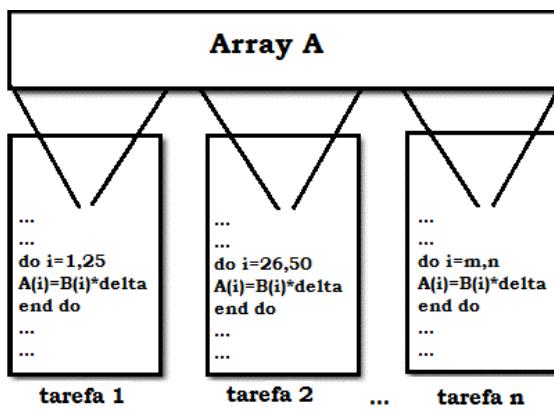


Figura 3.3: Modelo de Programação por Dados

A forma como a reorganização dos dados é efetuada não é definida pelo modelo, mas apenas o efeito final dos passos paralelos. Exemplos de implementações são Fortran 90/95, HPF (*High Performance Fortran*), MPI e OpenMP (*Open Multi-Processing*).

### 3.3.4 Thread e Multithreads

Na tentativa de diminuir o tempo gasto na criação e eliminação de subprocessos, bem como economizar recursos do sistema como um todo, foi introduzido o conceito de *thread*.

No ambiente *multithread*, múltiplas *threads* podem ser executadas dentro de um único processo. Cada *thread* possui seu próprio conjunto de registradores e pilha, porém compartilha o mesmo espaço de endereçamento, temporizadores e arquivos, de forma natural e eficiente com as demais *threads* do processo, como mostra a Figura 3.4.

Um exemplo de aplicação do modelo *Multithreads* é um processador de texto que tem uma *thread* para exibir gráficos, outra para ler os toques de tecla do usuário e uma terceira *thread* para executar a verificação ortográfica e gramatical em segundo plano [Silberschatz e Galvin 2004].

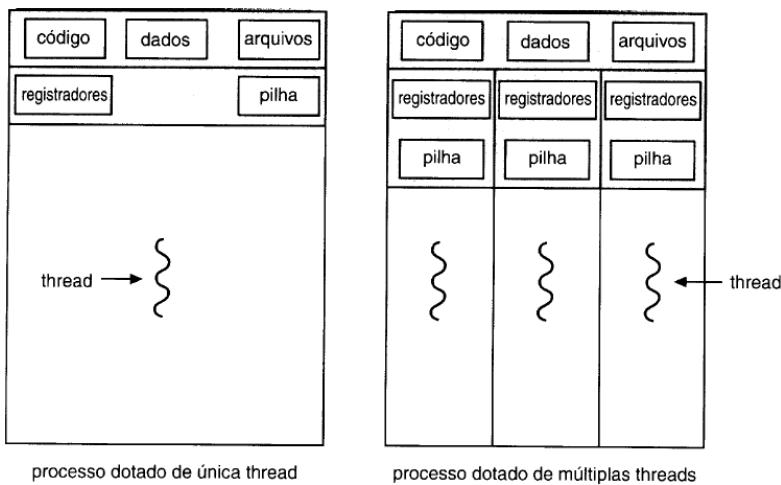


Figura 3.4: Modelo de Programação por Threads [Silberschatz e Galvin 2004]

### 3.3.5 MapReduce

O MapReduce [Ranger et al. 2007, Dean e Ghemawat 2008], criado pela Google em 2003, é um modelo de programação paralela para processamento largamente distribuído de grandes volumes de dados. Seu objetivo é facilitar a programação de aplicativos distribuídos com este perfil.

[Breshears 2009] define o MapReduce como um modelo de programação baseado em um meta-algoritmo, isto é, em padrões de resolução de problemas, como os conhecidos divisão-e-conquista e *backtracking*. O MapReduce utiliza um par de operações, baseado no paradigma funcional, denominado Map e Reduce, descrito na subseção 3.3.5.1. A Google construiu suas aplicações sobre o modelo MapReduce dentro de uma rede distribuída de milhares de servidores [Dean e Ghemawat 2008] e, assim, despertou a exploração desse método.

Esse tipo de processamento não era acessível fora de grandes centros de pesquisa ou empresas como a Google. Porém, baseando-se nos artigos liberados pela Google, Doug Cutting, criou em 2005 uma implementação *open source* em Java do MapReduce. O projeto chamado de Hadoop [Hadoop 2010, Venner 2009, White 2009] foi liderado pela Apache e apoiado pela Yahoo!. Sua adoção foi muito rápida e hoje ele é utilizado por grandes empresas, tais como o Facebook, Yahoo!, Twitter, Microsoft e IBM, que armazenam dados da ordem de PetaBytes ( $10^{15}$ ), e por laboratórios de Universidades como a *University of Maryland*, *Cornell University*, *University of Edinburg* e Unicamp. O Hadoop controla o sistema de busca da Yahoo! e define os anúncios exibidos ao lado dos resultados. Além disso, determina o que as pessoas visualizam na *homepage* da Yahoo!, ajuda a descobrir qual é a distância de conexão ou grau de separação entre usuários no Facebook, além de administrar as 40 bilhões de fotos armazenadas em seus servidores [Migliacci 2009].

### 3.3.5.1 Funcionamento do Modelo MapReduce

As funções Map e Reduce do MapReduce são definidas com relação a dados estruturados em pares (chave, valor). A ideia do Map consiste em tratar uma coleção de itens de dados, associar um valor a cada item na coleção e, assim, produzir uma coleção de pares (chave, valor). Essa operação deve ser completamente independente para cada elemento na coleção. A operação Reduce analisa todos os pares resultantes da operação Map e realiza uma computação de redução na coleção. O propósito é partir de uma coleção de itens de dados e retornar um valor derivado desses elementos. Em termos gerais, pode-se permitir que a operação de redução retorne zero, um ou qualquer número de resultados. Isso depende de qual operação de redução está sendo computada e dos dados de entrada obtidos da operação Map [Breshears 2009].

Tendo em vista, por exemplo, o problema de contagem do número de ocorrências de cada palavra em uma grande coleção de documentos, a função Map emite cada palavra mais a contagem das ocorrências associadas. A função Reduce, por sua vez, soma todas as contagens emitidas para uma determinada palavra na função Map. A Figura 3.5 esquematiza esse exemplo.

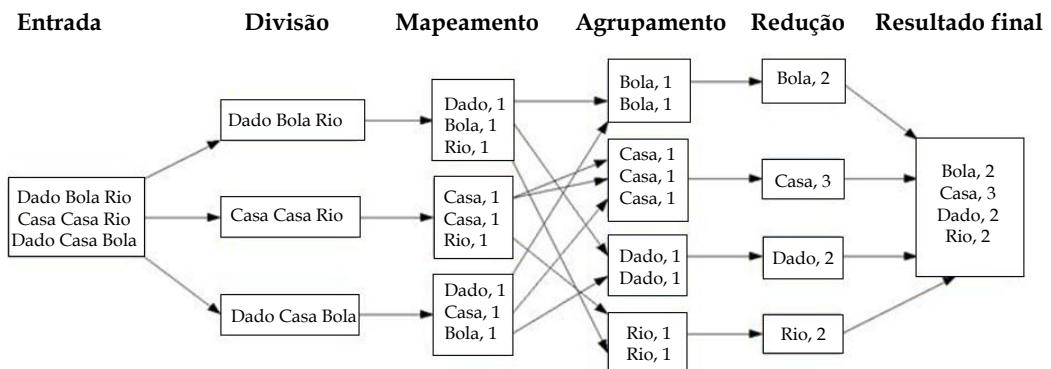


Figura 3.5: Contagem de ocorrências de palavras em documentos com o MapReduce

A Figura 3.6 mostra o fluxo geral de execução de uma operação MapReduce. Quando o programa do usuário chama a função MapReduce, a seguinte sequência de ações ocorre, como indicado pelas etiquetas da Figura 3.6:

1. A biblioteca MapReduce no programa do usuário primeiro divide os arquivos de entrada em  $M$  pedaços. Em seguida, iniciam-se muitas cópias do programa em um *cluster* de máquinas;
2. Uma das cópias do programa é especial: o mestre (*master*). O resto são trabalhadores (*escravos*, *slaves*) cujo trabalho é atribuído pelo mestre. Existem  $M$  tarefas Map e  $R$  tarefas Reduce a serem atribuídas. O mestre atribui aos trabalhadores ociosos uma tarefa Map ou uma tarefa Reduce;

3. Um trabalhador que recebe uma tarefa Map lê o conteúdo do fragmento de entrada correspondente. Ele analisa pares (chave, valor), a partir dos dados de entrada e encaminha cada par para a função Map definida pelo usuário. Os pares (chave, valor) intermediários, produzidos pela função Map, são colocados no *buffer* de memória;
4. Periodicamente, os pares colocados no *buffer* são gravados no disco local, divididos em regiões R pela função de particionamento. As localizações desses pares bufferizados no disco local são passadas de volta para o mestre, que é responsável pelo encaminhamento desses locais aos trabalhadores Reduce;
5. Quando um trabalhador Reduce é notificado pelo mestre sobre essas localizações, ele usa chamadas de procedimento remoto para ler os dados no *buffer*, a partir dos discos locais dos trabalhadores Map. Quando um trabalhador Reduce tiver lido todos os dados intermediários para sua partição, ela é ordenada pela chave intermediária para que todas as ocorrências da mesma chave sejam agrupadas. Se a quantidade de dados intermediários é muito grande para caber na memória, um tipo de ordenação externa é usado;
6. O trabalhador Reduce itera sobre os dados intermediários ordenados e, para cada chave intermediária única encontrada, passa a chave e o conjunto correspondente de valores intermediários para função Reduce do usuário. A saída da função Reduce é anexada a um arquivo de saída final para essa partição Reduce;
7. Quando todas as tarefas Map e Reduce são concluídas, o mestre acorda o programa do usuário. Neste ponto, a chamada MapReduce no programa do usuário retorna para o código do usuário.

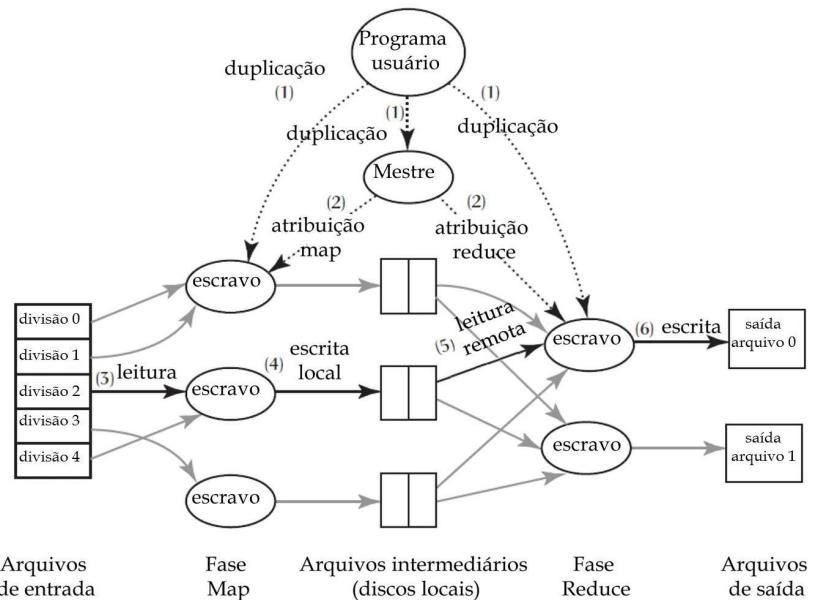


Figura 3.6: Execução do MapReduce [Dean e Ghemawat 2008]

### 3.4 Paralelização de Programas

Segundo [Rauber e Rünger 2010], a paralelização de um dado algoritmo ou programa é normalmente realizada na base do modelo de programação utilizado. No entanto, passos típicos podem ser identificados para realizar a paralelização. Para transformar computações sequenciais em um programa paralelo, o controle e as dependências de dados têm de ser levados em consideração para garantir que o programa paralelo produz os mesmos resultados que o programa sequencial para todos os possíveis valores de entrada. Seu objetivo principal geralmente é reduzir o tempo de execução do programa, tanto quanto possível, pelo uso de múltiplos processadores ou núcleos. Para realizar esta transformação de forma sistemática, podem ser identificados os seguintes passos, representados na Figura 3.7:

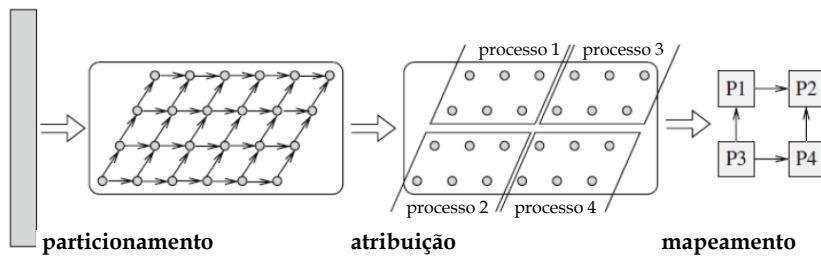


Figura 3.7: Passos para Paralelização de programas [Rauber e Rünger 2010]

- 1. Decomposição das computações:** as computações do algoritmo sequencial são divididas em tarefas e dependências entre as tarefas são determinadas. O objetivo da decomposição de tarefas é gerar tarefas suficientes para manter todos os núcleos ocupados em todos os momentos durante a execução do programa. O tempo de computação de uma tarefa também é conhecido por granularidade: tarefas com várias computações têm uma granularidade baixa e tarefas com apenas algumas computações apresentam granularidade alta. Se a granularidade da tarefa é muito alta, a sobrecarga de escalonamento e mapeamento é grande e constitui uma quantidade significativa do tempo total de execução. Assim, a etapa de decomposição deve encontrar um bom compromisso entre o número de tarefas e sua granularidade;
- 2. Atribuição de tarefas aos processos ou *threads*:** o número de processos ou *threads* não precisa ser necessariamente o mesmo que o número de processadores físicos ou núcleos, mas muitas vezes o mesmo número é usado. O objetivo principal dessa etapa é atribuir as tarefas de tal forma que uma boa carga de equilíbrio seja alcançada. A atribuição de tarefas aos processos ou *threads* é também chamada de *task assignment*. Para uma decomposição estática, a atribuição pode ser feita na fase de inicialização, no início do programa (assinalamento de tarefas estático). Mas o assinalamento de tarefas pode também ser feito durante a execução do programa (assinalamento de tarefas dinâmico);

3. **Mapeamento de processos ou *threads* para processos físicos ou núcleos:** no caso mais simples, cada processo ou *thread* é mapeado para um processador separado ou núcleo, também chamado unidade de execução. Se menos núcleos que *threads* estiverem disponíveis, múltiplas *threads* devem ser mapeadas para um único núcleo. Esse mapeamento pode ser feito pelo sistema operacional, mas também pode ser apoiado por comandos de programa. O objetivo principal dessa etapa é obter uma igual utilização dos processadores ou núcleos, enquanto a comunicação entre os processadores é mantida a menor possível.

Computadores *multicore* estão cada vez mais difundidos. Verifica-se que a cada geração dos processadores *multicore* alarga-se a lacuna entre o potencial do hardware e o desempenho que pode ser obtido com as aplicações atuais, por isso é fundamental aprender a escrever códigos paralelos, uma vez que essa é a única maneira de aproveitar os recursos oferecidos por tais processadores e conquistar melhorias de desempenho. Entretanto, a programação paralela exige o desenvolvimento de uma forma de pensar diferente da programação sequencial, introduz novos tipos de erros, desafios e problemas de desempenho. Os modelos de programação paralela são influenciados pela arquitetura e linguagem, compilador, ou pelas bibliotecas de execução. Além dos modelos tradicionais, há um novo modelo denominado MapReduce implementado de maneira *open source* pelo Hadoop. Esse modelo tem por objetivo facilitar a implementação de códigos paralelos e é utilizado para desenvolvimento das mais diversas aplicações, como a ordenação paralela.

## Capítulo 4

# Ordenação Paralela

A ordenação é utilizada para organizar elementos de uma sequência em determinada ordem, sendo um dos problemas fundamentais no estudo dos algoritmos [Knuth 1973]. Seu objetivo é facilitar a localização dos membros de um conjunto de dados. Assim sendo, é uma atividade fundamental e universalmente utilizada para a elaboração de algoritmos mais complexos [Quinn 1994].

Estima-se que a ordenação é responsável por mais de 80% de todo ciclo de processamento, seja apresentando resultados de consultas de banco de dados, compilando uma lista de investimentos de negócios com medidas de risco-recompensa associadas ou figurando na folha de pagamento das empresas [Breshears 2009]. Em consequência, é parte vital da computação e um dos problemas mais estudados na Ciência da Computação.

Muitos programas, tais como compiladores e sistemas operacionais, usam extensivamente a ordenação para lidar com tabelas e listas [Rajasekaran e Reif 1989]. A busca binária é outro exemplo de aplicação da ordenação, uma vez que necessita que a sequência de elementos esteja previamente ordenada. A ordenação também fornece uma forma fácil e rápida para solucionar o problema de verificação da duplicidade de elementos. Quando as operações são repetidas muitas vezes como, por exemplo, na busca contínua por elementos em uma massa de dados, é vantajoso ordenar a sequência primeiramente.

Os melhores algoritmos de ordenação sequenciais, tais como o *QuickSort* e o *HeapSort* [Aho e Hopcroft 1974], normalmente apresentam custo  $O(n \times \log n)$  para ordenar uma sequência de  $n$  chaves. O tempo aumenta acima do limite linear com o aumento do número de elementos. Deste modo, com o advento do processamento paralelo, foram desenvolvidas versões paralelas dos algoritmos sequenciais e criados novos algoritmos de ordenação paralela, com o intuito de diminuir consideravelmente o tempo de execução dos algoritmos sequenciais de ordenação [Rajasekaran e Reif 1989].

A ordenação paralela consiste no processo de uso de múltiplas unidades de processamento para ordenar coletivamente uma sequência desordenada. A sequência inicial é decomposta em subsequências disjuntas e cada uma é associada a uma única unidade de processamento. Na implementação de algoritmos de ordenação paralela, a questão fundamental é coletivamente ordenar os dados pertencentes a processos individuais, de tal forma que todas as unidades de processamento sejam utilizadas e, ao mesmo tempo, sejam minimi-

zados os custos de redistribuição de chaves entre os processadores [Kale e Solomonik 2010]. A complexidade da ordenação paralela tem sido medida em termos do número de processadores usados e do número de operações realizadas [Leighton 1985].

A ordenação tem uma importância adicional para os desenvolvedores de algoritmos paralelos. Ela é frequentemente utilizada para realizar permutações de dados em computadores de memória distribuída. Estas operações de movimentos de dados podem ser usadas para resolver problemas de teoria dos grafos, computação geométrica e processamento de imagem em tempo próximo ao ótimo [Quinn 1994]. A ordenação paralela também pode ser utilizada no apoio de aplicações de alto desempenho científico e industriais, tais como em simulações cosmológicas e em portais de viagens. O uso da ordenação em portais de viagens pode ser exemplificado pela consulta de muitos vôos diferentes no banco de dados de cada companhia aérea e exibição dos mais baratos em todas as transportadoras aéreas para o usuário.

Os algoritmos paralelos para ordenação têm sido estudados desde o começo da computação paralela. Um dos primeiros métodos propostos foi o *Bitonic Sort Network*, em 1968 [Batcher 1968]. Desde então, muitos algoritmos de ordenação paralela foram propostos para arquiteturas vetoriais, multiprocessadores e multicamputadores [Hennessy e Patterson 2007, Blelloch 1990, Leighton 1992]. Na década de 80, foi desenvolvido o algoritmo *FlashSort* [Reif e Valiant 1987] que utiliza uma sofisticada técnica de amostragem ao acaso para formar um conjunto de divisão. Uma versão similar a ele é o *SampleSort* [Huang e Chow 1983] que distribui o divisor definido para cada processador. Outros notáveis algoritmos de ordenação são as versões paralelas do *RadixSort*, *Quick-Sort* [Blelloch 1990, Leighton 1992, Breshears 2009] e *MergeSort* [Cole 1988], assim como os novos algoritmos de ordenação paralelos como o PSRS (*Parallel Sorting by Regular Sampling*) [Shi e Schaeffer 1992].

Devido ao grande número de algoritmos de ordenação paralela e à variedade de arquiteturas paralelas, é uma tarefa difícil selecionar o melhor algoritmo para uma determinada máquina e tipo de problema. A principal razão dessa dificuldade é o fato de não existir um modelo teórico conhecido que possa ser aplicado para prever com precisão o desempenho de um algoritmo em arquiteturas diferentes. Dessa forma, estudos experimentais reconhecem uma importância crescente na avaliação e seleção de algoritmos apropriados para multiprocessadores. Uma série de estudos nesta direção são encontrados na literatura [Blelloch, G. et al 1991, Li e Sevcik 1994]. No entanto, mais estudos são necessários antes que se possa indicar um determinado algoritmo como recomendado para uma determinada máquina com algum grau de confiança.

Em geral, quanto mais aplicações paralelas vão sendo desenvolvidas hoje, mais inovadores devem ser os algoritmos de ordenação paralela concebidos para suportá-las. Como algoritmos de ordenação são sensíveis à distribuição inicial das chaves, a compreensão do contexto e das necessidades da aplicação final é importante na concepção e implementação

de algoritmos de ordenação paralela [Kale e Solomonik 2010].

#### 4.1 Limites da Ordenação Paralela

Diversas soluções de ordenação podem ser consideradas ao implementar um algoritmo de ordenação paralela em um modelo de programação paralela. Cada uma delas atende a uma aplicação paralela e/ou a uma particular plataforma ou arquitetura de máquina. Dessa forma, o programador precisa considerar diversas decisões de projeto, com análise cuidadosa das características da aplicação na implementação dos algoritmos de ordenação paralelos. As principais questões a serem analisadas são [Kale e Solomonik 2010]:

- **Habilidade de explorar distribuições iniciais parcialmente ordenadas:** quando a ordenação é feita periodicamente na mesma sequência, a ordem da sequência é normalmente somente perturbada. Alguns algoritmos de ordenação paralela podem tirar vantagem deste cenário e realizar menos trabalho ou menos movimento de dados;
- **Movimento de dados:** o custo de migração de dados entre processadores em um sistema de memória distribuída pode dominar o custo de execução global e, assim, pode ser um ponto de contenção de escalabilidade fundamental. Alguns algoritmos de ordenação paralela atingem o mínimo limite de comunicação, porém tendem a apresentar maior latência e algum esforço para manter o balanceamento de carga de forma eficiente;
- **Balanceamento de carga:** a ordenação paralela deve assegurar o balanceamento de carga na organização dos dados, uma vez que praticamente todas as aplicações paralelas requerem que os dados sejam distribuídos uniformemente por todos processadores. Sua importância é imensa, visto que o tempo de execução da aplicação é tipicamente limitado pelo tempo de execução local do processador mais sobrecarregado. Qualquer algoritmo de ordenação paralelo moderno tem um mecanismo de manutenção do equilíbrio de carga, sendo a qualidade desse mecanismo um diferencial em relação aos demais algoritmos de ordenação;
- **Latência de comunicação:** consiste no tempo médio necessário para enviar uma mensagem de um processador para outro. A latência não é um problema para máquinas com um pequeno número de processadores. No entanto, se torna muito importante para máquinas distribuídas em larga escala. Especialmente se a relação entre o número de dados a ser ordenado e o número de processadores é pequena, a latência de movimento de dados pode ter um enorme peso sobre a escalabilidade e o desempenho de um algoritmo de ordenação paralela. O limite de latência mínima pode ser alcançado apenas através de combinação de mensagens, o que significa que não se pode alcançar ao mesmo tempo mínima latência e mínima movimentação de dados;

- **Largura de banda de comunicação suplementar:** largura de banda é a taxa de dados agregados em que os processadores enviam e recebem as chaves. Refere-se à largura de banda obtida pelo algoritmo, ao invés da largura de banda disponível na rede de interconexão da máquina. Todo algoritmo de ordenação paralelo tipicamente realiza alguma comunicação adicional para atingir balanceamento de carga ou organizar o movimento de dados. Essa comunicação suplementar frequentemente se torna o principal gargalo de escalabilidade ao ordenar conjuntos de dados menores sobre uma grande quantidade de processadores;
- **Sobreposição de comunicação com a computação:** a sobreposição permite que enquanto um processo realiza alguma operação de entrada e saída, geralmente de natureza bloqueante, um segundo processo possa utilizar o processador para a realização de algum cálculo, o que reduz o tempo total de execução e evita que os recursos fiquem ociosos durante todo o intervalo de tempo necessário para a transmissão da carga de trabalho.

## 4.2 Descrição de Algoritmos de Ordenação

Nesta seção são descritos importantes algoritmos de ordenação e os conceitos envolvidos em suas versões paralelizadas.

### 4.2.1 BubbleSort

*BubbleSort*, também chamado método da bolha, é um dos mais simples algoritmos de ordenação. Ele utiliza a estratégia de “comparação e troca” sobre os  $n$  elementos de um vetor que se deseja ordenar. A ideia é comparar os  $n$  elementos dois a dois e trocá-los de posição se estiverem fora de ordem. Dessa forma, os maiores valores são deslocados para o final da lista. Essa movimentação lembra a forma como as bolhas em um tanque de água procuram seu próprio nível, o que origina o nome do algoritmo.

A complexidade da versão sequencial desse algoritmo é de ordem quadrática ( $O(n^2)$ ). Por isso, ele não é recomendado para programas que necessitam de velocidade e operam com quantidade elevada de dados. A Figura 4.1 mostra as comparações realizadas nas duas primeiras iterações desse algoritmo para ordenar um vetor de dez chaves inteiros. Nos sucessivos vetores apresentados, são refletidos os resultados da movimentação de dados.

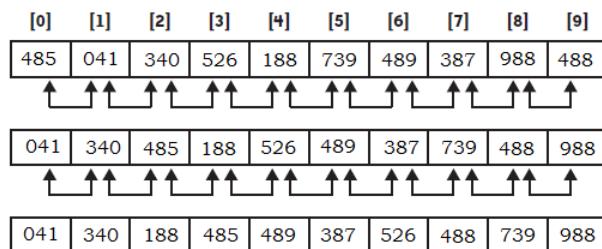


Figura 4.1: Comparações realizadas no *BubbleSort* [Breshears 2009]

Uma abordagem paralela desse algoritmo consiste na decomposição de dados do vetor a ser ordenado por meio da técnica *pipelining*. Ela permite a execução de várias fases simultaneamente, pois cada fase só faz uma comparação para cada par de posições. Quando a primeira fase se encontra na comparação entre as posições  $x_2$  e  $x_3$ , a segunda fase já pode fazer a comparação entre as posições  $x_0$  e  $x_1$ . A terceira fase terá início quando a segunda fase estiver comparando as posições  $x_2$  e  $x_3$ , e a primeira fase, por sua vez, estiver realizando a comparação das posições  $x_4$  e  $x_5$ , como mostra a Figura 4.2.

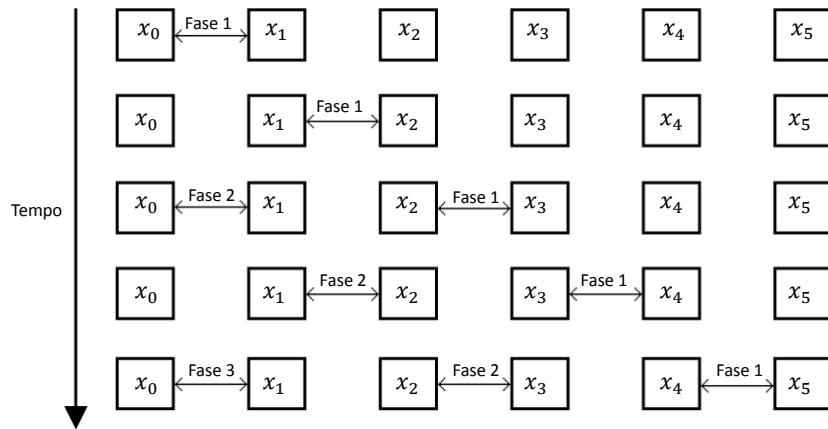


Figura 4.2: Bubblesort com *pipelining*

Essa versão paralela do *BubbleSort* tem complexidade  $O(n)$  e inclui uma pequena discussão da complexidade da ordenação em paralelo.

#### 4.2.2 QuickSort

O algoritmo *QuickSort* é um método de ordenação muito rápido e eficiente, inventado por Hoare em 1960 e publicado em 1962, após uma série de refinamentos [Shustek 2009]. Ele aplica a abordagem “dividir para conquistar” pelo recursivo particionamento de uma sequência utilizando um elemento pivô. O primeiro passo é a escolha de um elemento da lista como pivô. Feito isso, a lista é rearranjada de forma que todos os elementos maiores do que o pivô fiquem de um dos lados do pivô e todos os elementos menores fiquem do outro lado. Esse processo é repetido recursivamente para cada sublista e, no final, tem-se uma lista ordenada. A complexidade do *QuickSort* é  $O(n \times \log n)$ .

A Figura 4.3 mostra três etapas de particionamento de um vetor de dez chaves inteiras. A primeira linha mostra os dados originais; a segunda mostra os dados após o vetor ter sido particionado; e a terceira mostra como cada uma das duas partições, a partir da segunda linha, seria particionada. As caixas encapsulam as partes do vetor que precisam ser ordenadas.

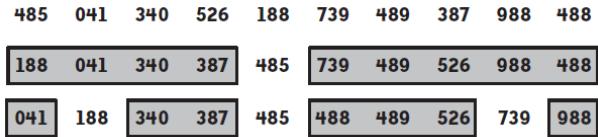


Figura 4.3: Partições do *QuickSort* [Breshears 2009]

A paralelização do *QuickSort* normalmente é alcançada pelo uso de pivôs para recursivamente particionar o conjunto de processadores que interagem. O balanceamento de carga é mantido pela semântica da seleção do pivô [Kale e Solomonik 2010]. Uma das vantagens deste algoritmo em relação a outros algoritmos em paralelo é o fato de não ser necessária a sua sincronização. Um processo é criado por cada sublista gerada e trabalha apenas com essa lista, não comunicando com os outros processos. Em relação à complexidade dessa versão paralela do *QuickSort*, tendo-se  $p$  processadores, pode-se dividir a lista de  $n$  elementos em  $p$  sublistas em um tempo  $O(n)$ . Após isso, a ordenação será concluída em um tempo  $O((n/p)\log(n/p))$ .

#### 4.2.3 RadixSort

*RadixSort* é um algoritmo de ordenação por distribuição que utiliza a estrutura binária das chaves para ordenar os dados. É possível utilizá-lo apenas se a representação binária das chaves pode ser interpretada como um inteiro sem sinal. A comparação entre partes de chaves, ao invés da chave como um todo, confere ao *RadixSort* uma complexidade assintótica de tempo linear, em oposição à complexidade polinomial dos outros (comparatroc) algoritmos de ordenação, descritos nas subseções 4.2.1 e 4.2.2. A complexidade de tempo do *RadixSort* é  $O(nk)$ , onde  $k$  é o número de dígitos usados nas partições *radix* e  $n$  o número de elementos a ser ordenado.

Existem dois métodos de processamento das chaves binárias que definem dois tipos distintos de RadixSort [Blelloch, G. et al 1991, Breshears 2009]. O tratamento direcionado dos bits mais significativos aos menos significativos (da esquerda para a direita) define o *Radix Exchange Sort*. A Figura 4.4 ilustra o particionamento de chaves nesse tipo de *RadixSort*. Como ela mostra, primeiramente considera-se apenas o primeiro bit mais significativo da sequência a ser ordenada, que então é dividida em dois conjuntos, um formado pelas chaves cujo bit mais significativo é 0 e outro pelas chaves que tem 1 como bit mais significativo. O processo se repete para o segundo bit mais significativo e formam-se novos conjuntos ordenados.

Ordem original	485	041	340	526	188	739	489	387	988	488
	0111100101	0000101001	0101010100	01000001110	0010111100	1011100011	0111101001	0110000011	1111011100	0111101000
Uso de 1 bit										
	chaves “0”					chaves “1”				
Uso de 2 bits										
	chaves “00”			chaves “01”			chaves “10” chaves “11”			
Chaves de inteiros	188	041	340	526	485	488	489	387	739	988

 Figura 4.4: Duas fases do *Radix Exchange Sort* [Breshears 2009]

Já o processamento de bits do menos ao mais significativo (direita para a esquerda), é conhecido como *Straight Radixsort*. A Figura 4.5 mostra a aplicação desse tipo de ordenação para chaves de três dígitos, a uma taxa de uma passagem por dígito. A barra (|) é colocada à esquerda do dígito a ser ordenado. Como representado, ordena-se primeiro o bit menos significativo (última coluna da tabela) e depois os mais significativos até ordenar toda a tabela.

Lista Original	Após 1º passo	Após 2º passo	Após 3º passo
485	34 0	5 26	041
041	04 1	7 39	188
340	48 5	3 40	340
526	52 6	0 41	387
188	38 7	4 85	485
739	18 8	3 87	488
489	98 8	1 88	489
387	48 8	9 88	526
988	73 9	4 88	739
488	48 9	4 89	988

 Figura 4.5: *Straight Radixsort* usando dígitos decimais [Breshears 2009]

A versão paralela do *RadixSort* depende de chaves de tamanho limitado e funciona melhor quando o tamanho das chaves é pequeno (por exemplo, 32 bits e 64 bits). Ela associa cada partição *radix* a um processador. O conjunto de chaves é dividido nessas partições. Em cada iteração, o processador determina a posição correta do elemento dentro da partição. Ao terminar de ordenar a partição, cada processador envia ou copia cada chave para a partição apropriada. Uma vez que todos os processadores copiaram os elementos de sua partição para novas partições (se necessário), as próximas iterações prosseguem usando os próximos bits da chave. A complexidade de tempo dessa versão paralela é  $O(kn/p)$  onde  $k$  é o número de dígitos usados no *radix*,  $n$  o número de elementos e  $p$  o de processadores [Kale e Solomonik 2010].

### 4.3 Ordenação Externa

A ordenação externa deve ser aplicada em conjuntos de dados compostos por um número de registros maior do que a memória interna do computador pode armazenar. Os métodos de ordenação externa são muito diferentes dos métodos de ordenação interna, apesar de, em ambos os casos, o problema ser o mesmo: rearranjar os registros de um arquivo em ordem ascendente ou descendente. Isso ocorre, pois na ordenação externa as estruturas de dados têm que levar em conta o fato de que os dados estão armazenados em unidades de memória externa, tais como fitas e discos magnéticos, relativamente mais lentas do que a memória principal em dois aspectos: velocidade de acesso e tipo de acesso [Ziviani 2007, Knuth 1973].

Os dados são armazenados na memória externa como um arquivo sequencial, em que apenas um registro pode ser acessado em um dado momento. Essa forte restrição torna os métodos de ordenação interna inadequados para a ordenação externa. Surge então, a necessidade de se utilizar técnicas de ordenação completamente diferentes [Ziviani 2007, Knuth 1973].

Na ordenação externa, os itens que não estão na memória principal devem ser buscados em memória secundária e trazidos para a memória principal, para assim serem comparados. Esse processo se repete inúmeras vezes, o que o torna lento, uma vez que os processadores ficam grande parte do tempo ociosos à espera da chegada dos dados à memória principal para serem processados. Por esse motivo, a grande ênfase de um método de ordenação externa deve ser na minimização do número de vezes que cada item é transferido entre a memória interna e a memória externa. Além disso, cada transferência deve ser realizada de forma tão eficiente quanto as características dos equipamentos disponíveis permitam [Ziviani 2007].

O método de ordenação por intercalação é considerado o método mais importante de ordenação externa. Intercalar consiste em combinar dois ou mais blocos ordenados em um único bloco ordenado. A intercalação é utilizada como uma operação auxiliar na ordenação [Ziviani 2007].

A estratégia geral dos métodos de ordenação externa é a seguinte [Ziviani 2007]:

1. Quebre o arquivo em blocos do tamanho da memória interna disponível;
2. Ordene cada bloco na memória interna;
3. Intercale os blocos ordenados, por meio da realização de várias leituras sobre o arquivo;
4. A cada leitura são criados blocos ordenados cada vez maiores, até que todo o arquivo esteja ordenado.

Os algoritmos para ordenação externa devem reduzir o número de passadas (leituras) sobre o arquivo. Como a maior parte do custo se concentra nas operações de entrada e saída de dados da memória interna, uma boa medida de complexidade de um algoritmo de

ordenação por intercalação corresponde ao número de vezes que um dado é lido ou escrito na memória auxiliar [Ziviani 2007].

Existem duas situações em que as técnicas de ordenação externa não são capazes de ordenar grandes quantidades de dados de forma eficiente, isto é, em tempo viável: quando a quantidade de memória física disponível é pequena ou quando a massa de dados é tão grande que a redução de acessos à disco não é suficiente para diminuir significativamente o tempo de processamento. Nesses casos, a programação paralela pode ser utilizada para otimizar o processo. Dessa forma, cada processador fica responsável por ordenar uma parte dos dados, que quando fundidos por um computador mestre, formam o arquivo completamente ordenado. Um algoritmo paralelo com esse princípio de ordenação é descrito na seção 4.5.

#### 4.4 Ordenação na Plataforma Hadoop

O *Sort* é um dos mais conhecidos *benchmarks* de ordenação de dados para Hadoop, definido por Jim Gray em 1998 [Gray 1998]. Consiste em um conjunto de muitos *benchmarks* relacionados, cada um com suas regras. Todos eles medem os tempos para ordenar diferentes números de registros de 100 Bytes. Os primeiros 10 Bytes de um registro são a chave e o restante o valor do elemento a ser ordenado. No benchmark *Sort* as entradas e saídas devem ser sequências de arquivos onde as chaves e os valores são gravados em Byte.

As principais categorias dos benchmarks *Sort* são a *MinuteSort* e a *GraySort*. A categoria *MinuteSort* deve ordenar a maior quantidade dos dados em um minuto e a *GraySort* deve ordenar mais que 100 TeraBytes em pelo menos uma hora [White 2009].

Os *benchmarks* relacionados ao *Sort* possuem uma aplicação MapReduce, que realiza uma ordenação parcial dos dados de entrada, e são constituídos de três passos:

1. **Geração de dados:** gera dados aleatórios a serem ordenados;
2. **Ordenação dos dados:** ordena os dados gerados pelo passo 1;
3. **Validação de dados:** valida os resultados obtidos pela ordenação dos dados realizada no passo 2.

O *TeraSort* é outra aplicação de destaque para ordenação de dados com Hadoop, criada por Owen O' Malley [O'Malley e Murthy 2009], com o intuito de participar da competição de ordenação chamada *GraySort* [Gray 1998]. Em 2009, o *TeraSort* foi o campeão dessa competição em duas categorias, ao ordenar 500 GB em 59 segundos (*MinuteSort*) utilizando um *cluster* com 1.406 nodos e 100 TB em 173 minutos (*GraySort*) em um *cluster* com 3.452 nodos. A escalabilidade da solução foi provada pela ordenação de 1 PB em 975 minutos (equivalente a 16,25 horas) em 3.658 nodos.

#### 4.5 Ordenação por Amostragem

O algoritmo de Ordenação por Amostragem [White 2009, Venner 2009] é um método de ordenação externa para ordenar grandes quantidades de dados. A ideia desse algoritmo

é dividir um conjunto de dados a ser ordenado em subconjuntos (partições) de forma que os elementos da partição à esquerda sejam menores do que os elementos à direita da partição, isto é, todas as chaves da partição  $i$  são menores que as chaves da partição  $i + 1$ . Cada uma das partições é designada a um processador para ordenação (ordenação interna) e, assim, é produzido um conjunto de arquivos ordenados que, se concatenados, formam um arquivo globalmente ordenado.

A divisão dos dados em partições (particionamento) é feita por meio de uma estratégia de amostragem. Essa estratégia baseia-se na análise de um subconjunto de dados, denominado amostra, ao invés de todo o conjunto, para estimar a distribuição de chaves e, assim, construir partições balanceadas.

Existem três tipos de estratégias de amostragem: *SplitSampler*, *IntervalSampler* e *RandomSampler*. O *SplitSampler* seleciona apenas os  $n$  primeiros registros do arquivo para formar a amostra. Já o *IntervalSampler* forma a amostra por meio da escolha de chaves em intervalos regulares no arquivo. No *RandomSampler*, a amostra é constituída por chaves selecionadas aleatoriamente no conjunto.

Após a formação das amostras, são definidos os limites para as partições, isto é, o intervalo de valores compreendido por cada partição. A quantidade de limites amostrados ( $q_{la}$ ) é determinada pelo equação:  $q_{la} = n_p - 1$ , onde  $n_p$  é o número de partições ( $n_p = n_c \times n_{mag}$ , sendo  $n_c$  o número de *cores* e  $n_{mag}$  o número de máquinas). As informações das partições são armazenadas em um arquivo e transmitidas para as máquinas participantes por meio de cache distribuído.

A determinação e designação de cada elemento do conjunto para a partição apropriada é feita por um dos dois tipos de pesquisa seguintes [Venner 2009]:

- Pesquisa binária: o algoritmo de Ordenação por Amostragem constrói e realiza pesquisa em uma árvore binária, quando os dados de entrada não estão em formato binário. A árvore binária é uma estrutura de dados baseada em nós, na qual todos os nós da subárvore esquerda possuem um valor numérico inferior ao nó raiz e todos os nós da subárvore direita possuem um valor superior ao nó raiz, ou vice-versa [Ziviani 2007];
- Pesquisa digital: quando os dados de entrada encontram-se em representação binária, o algoritmo de Ordenação por Amostragem constrói e realiza busca em uma árvore de prefixo binária (*trie*). A árvore *trie* é uma estrutura de dados do tipo árvore ordenada, que pode ser usada para armazenar conjuntos de dados em que as chaves são normalmente cadeias de caracteres. Ao contrário de uma árvore de busca binária, nenhum nó nessa árvore armazena a chave associada a ele. A chave, nesse caso, é determinada pela sua posição na árvore. As principais vantagens da *trie* sobre a árvore de busca binária consistem no fato da busca ser mais rápida e ser necessário menos espaço para um grande número de cadeias curtas, uma vez que as chaves não são armazenadas de forma explícita [Ziviani 2007].

O melhor tipo de estratégia de amostragem é determinado pela entrada de dados. Para arquivos quase ordenados, o *SplitSampler* não é recomendado, uma vez que não seleciona as chaves de todo o conjunto e, assim, não produz bons resultados. Nesse caso, a melhor escolha é o *IntervalSampler* pelo fato de selecionar chaves que representam melhor a distribuição do conjunto. O *RandomSampler* é considerado um bom amostrador de propósito geral [White 2009], por isso será utilizado neste trabalho.

O *RandomSampler* apresenta os seguintes parâmetros de balanceamento: probabilidade de escolha de uma chave, o número máximo de amostras a serem selecionadas para realizar a amostragem e o número máximo de partições que podem ser utilizadas. Ele termina quando algum dos parâmetros é atingido.

A Figura 4.6 apresenta um esquema da estrutura de funcionamento do algoritmo, quando implementado em MapReduce no ambiente Hadoop. O algoritmo pode ser dividido em duas fases: Map e Reduce.

Na fase Map, os arquivos de entrada são lidos, ocorre a formação do vetor inicial e dos pares (chave, valor) dos valores lidos (passos 1.1 e 1.2). Depois disso, os dados são divididos em partições cujo número é determinado pelo número de máquinas e seus núcleos de processamento (passo 1.3). No exemplo apresentado, são utilizados dois núcleos de processamento (*cores*). Portanto, existem duas partições e, dessa forma, deve ser escolhido apenas um limite para as partições. Esse limite é definido por meio da seleção do elemento que melhor representa a distribuição de chaves na amostra formada pela estratégia *RandomSampler*. Para esse exemplo o limite escolhido foi o número 13, o que implica que em uma partição estarão valores menores e iguais a 13 e, na outra, valores maiores que 13. Por meio de cache distribuído, as informações das partições são transmitidas para as máquinas participantes e os dados particionados. Cada partição é então atribuída a uma tarefa Reduce (processador) diferente.

Na fase Reduce, os dados são ordenados localmente (passo 2.1), ou seja, em cada processador os dados são ordenados na memória interna. Para essa ordenação, avalia-se a profundidade da árvore de recursão e, se ela for baixa, utiliza-se o algoritmo *QuickSort*. Caso contrário, o *HeapSort* é utilizado [White 2009]. Após isso, os dados retornam para a máquina *master*, onde são concatenados, formando o vetor ordenado.

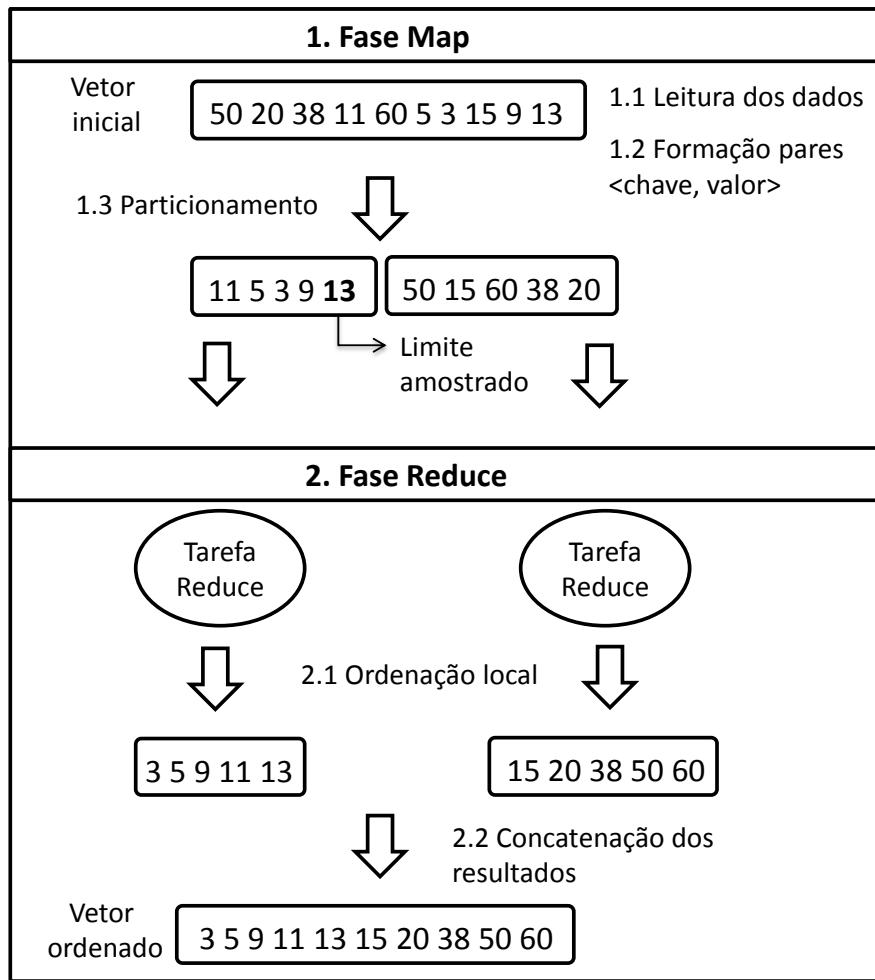


Figura 4.6: Funcionamento do algoritmo de Ordenação por Amostragem

O balanceamento das partições, ou seja, a formação de partições aproximadamente iguais no tamanho é muito importante para o algoritmo de Ordenação por Amostragem, pois evita que os tempos de ordenação sejam dominados por um único processador. Em outras palavras, o equilíbrio das partições reduz a possibilidade de que um processador esteja ocioso, enquanto outro processador está sobrecarregado, situação que comprometeria o desempenho do algoritmo [White 2009].

## Capítulo 5

# Metodologia e Planejamento dos Experimentos

Este trabalho iniciou-se com uma ampla pesquisa bibliográfica de material de apoio e estudo da computação paralela, ordenação paralela, dos principais conceitos e funcionamento da plataforma Hadoop, assim como do modelo MapReduce em que ela se baseia. A partir dessa pesquisa, foi possível entender os fundamentos do paralelismo, sua importância, bem como sua possível aplicação em métodos de ordenação, com uso do modelo MapReduce.

Para compreender o funcionamento da plataforma Hadoop e do MapReduce, o ambiente Hadoop foi instalado primeiramente em uma única máquina *single-node* e, posteriormente, foi configurada sua execução em *cluster*, isto é, em várias máquinas *multi-node*. Esse procedimento foi adotado, pois é mais simples identificar os problemas que possam ocorrer em cada máquina, quando essas estão configuradas isoladamente.

Em cada uma das instalações foram realizados testes para verificar se o *cluster* estava funcionando corretamente, por meio de execuções iniciais dos programas *WordCount* e *Pi*, que acompanham a instalação do Hadoop. O programa *WordCount* conta as ocorrências de palavras em diferentes documentos e o programa *Pi* realiza a estimativa do número  $\pi$  ( $\pi = 3,141592654$ ) com várias casas decimais. Após isso, iniciou-se a execução dos primeiros testes de ordenação, utilizando-se os *benchmarks* *TeraSort* e *Sort*, descritos na seção 4.4.

A etapa seguinte consistiu no estudo de algoritmos de ordenação, de suas versões paralelas e da ordenação no ambiente Hadoop. Os algoritmos estudados foram o *BubbleSort*, *QuickSort*, *RadixSort* e Ordenação por Amostragem, descritos no capítulo 4. A ordenação no ambiente Hadoop foi investigada com base nos algoritmos dos *benchmarks* *TeraSort* e *Sort*. Por meio desse estudo, definiu-se a implementação do algoritmo de Ordenação por Amostragem, pois o mesmo utiliza ordenação externa e interna, além de uma estratégia diferente dos demais, denominada amostragem, para criação de partições balanceadas.

O algoritmo de Ordenação por Amostragem foi implementado na linguagem Java, de acordo com o modelo MapReduce, no ambiente Hadoop. Após os testes do *cluster*, foram realizados testes de verificação e validação do algoritmo com a finalidade de verificar sua correção e consistência. Além desses, foram feitos testes para verificar a estabilidade do ambiente, comportamento com diferentes conjuntos de dados, quantidade de dados e máquinas. Os testes são descritos na seção 5.2.

## 5.1 Ambiente de Desenvolvimento e Testes

Os testes deste trabalho, descritos na seção 5.2, foram realizados utilizando o *cluster* do Laboratório de Redes e Sistemas (LABORES) do Departamento de Computação (DECOM). O *cluster* é formado por cinco máquinas Dell Optiplex 380 com as seguintes características:

- Sistema operacional Linux Ubunbu 10.04 32 bits;
- Processador Intel Core 2 Duo de 3,0 GHz;
- Memória RAM de 4 GB;
- Disco rígido SATA de 500 GB 7200 RPM;
- Placa de rede Gigabit Ethernet;
- Sun Java JDK 1.6.0\_19.0-b09;
- Hadoop 0.20.2.

As características do ambiente delimitaram e limitaram os tipos de testes realizados. É importante ressaltar que, comparado com a literatura, o Hadoop é altamente escalável. Dados das competições anuais de ordenação utilizando o Hadoop mostram que quantidades cada vez maiores de dados são ordenadas com o *framework* de forma eficiente e consistente, em *clusters* compostos por grandes quantidades de máquinas [Gray 1998]. Em 2009, por exemplo, a aplicação *TeraSort* ordenou um PetaByte em 975 minutos (16,25 horas) em 3.658 máquinas [O’Malley e Murthy 2009]. O Facebook utiliza dois *clusters* para executar aplicações com Hadoop, sendo cada um formado por oito *cores* e doze TeraBytes de armazenamento. Um deles é composto por 1.100 máquinas e tem cerca de doze PetaBytes de armazenamento bruto e o outro é formado por 300 máquinas e possui em torno de três Petabytes de armazenamento bruto [Hadoop 2010].

## 5.2 Planejamento de Testes

Nesta seção são descritos e discutidos os tipos de testes realizados no trabalho.

### 5.2.1 Testes Utilizando *Benchmarks* de Ordenação

Os *benchmarks* *TeraSort* e *Sort* foram utilizados para realizar os primeiros testes de ordenação no *cluster*. Os objetivos desses testes consistiam em entender e analisar o funcionamento da ordenação no ambiente Hadoop, compreender as estratégias de implementação adotadas pelos algoritmos dos dois *benchmarks* que mais se destacam nas competições de ordenação no Hadoop, bem como avaliar a capacidade do ambiente de testes.

Os testes do *benchmark* *TeraSort* foram realizados utilizando-se duas máquinas do *cluster* descrito na seção 5.1. Ao executar o algoritmo *TeraSort* para os dados gerados, foi obtido um arquivo com os registros dos dois arquivos de entrada ordenados em ordem

crescente de acordo com a chave. Após a execução do *TeraSort*, realizou-se a validação dos dados ordenados por meio do *TeraValidate*, que é um programa que verifica se os dados foram ordenados corretamente.

Os testes do *benchmark Sort* foram realizados utilizando-se quatro máquinas do *cluster* descrito na seção 5.1. Os dados gerados foram lidos e ordenados em ordem crescente pelo algoritmo *Sort*.

### 5.2.2 Testes de Verificação

Testes de verificação e validação foram realizados para o algoritmo de Ordenação por Amostragem com o intuito de analisar se o programa de ordenação estava funcionando corretamente para diferentes entradas, quantidade de dados e máquinas.

Para isso, foram utilizados como dados de entrada valores inteiros aleatórios homogêneos (próximos), como o conjunto  $\{21, 18, 20, 11, 10, 12, \dots\}$  ou viciados (distantes), como o conjunto  $\{1000, 15, 13, 10, 9, 5, \dots\}$ . A quantidade de dados para esses dois tipos de conjuntos variaram em valores pares e ímpares, como 20 e 1000 e 21 e 1001. As combinações de valores possíveis foram então executadas utilizando-se duas e três máquinas do *cluster*.

### 5.2.3 Testes de Estabilidade do Sistema

Após os testes de verificação, foram realizados testes para avaliar a estabilidade do sistema, isto é, a variabilidade do tempo de resposta na execução do algoritmo.

Para isso, o algoritmo foi executado dez vezes, em quatro máquinas, para um arquivo com  $10^6$  chaves inteiras aleatórias com os parâmetros de balanceamento apresentados na Tabela 5.1.

### 5.2.4 Testes Variando o Conjunto de Dados

O algoritmo de Ordenação por Amostragem foi também executado para dez conjuntos de dados diferentes, ou seja, diferentes conjuntos de valores inteiros aleatórios, todos com a mesma quantidade de dados ( $10^6$ ). Para cada conjunto, o algoritmo foi executado dez vezes em quatro máquinas, com os parâmetros de balanceamento apresentados na Tabela 5.1. O objetivo desse teste foi avaliar a influência dos dados aleatórios gerados para ordenação nos resultados relativos ao desempenho do algoritmo.

### 5.2.5 Testes Variando a Quantidade de Dados

O tempo de ordenação do algoritmo de Ordenação por Amostragem foi analisado para diferentes quantidades de dados ( $10^6$  a  $10^{10}$  inteiros aleatórios) em quatro máquinas. Para cada quantidade de dados, o algoritmo foi executado três vezes com os parâmetros de balanceamento apresentados na Tabela 5.1. Neste teste, buscou-se verificar os tempos obtidos para diferentes quantidades de dados e o aumento de tempo provocado à medida que a quantidade de dados a ser ordenada aumenta. O objetivo é avaliar a complexidade do algoritmo quando o conjunto a ser ordenado aumenta.

### 5.2.6 Testes Variando a Quantidade de Máquinas

O algoritmo de Ordenação por Amostragem também foi executado considerando o mesmo conjunto de dados ( $10^8$  inteiros aleatórios) em diferentes quantidades de máquinas (duas a cinco máquinas). Para cada quantidade de máquinas, o algoritmo foi executado três vezes com os parâmetros de balanceamento apresentados na Tabela 5.1. A finalidade desse teste residiu em analisar a escalabilidade do algoritmo, ou seja, a diminuição no tempo de ordenação (melhora no desempenho) resultante do aumento do número de máquinas participantes do processo.

## 5.3 Geração da Carga de Trabalho

Nesta seção é descrito como a carga de trabalho, ou seja, os dados a serem ordenados foram gerados para os testes utilizando os *benchmarks* de ordenação e para os testes do algoritmo de Ordenação por Amostragem.

### 5.3.1 Benchmarks de Ordenação

Os dados para ordenação utilizando o *benchmark* *TeraSort* foram gerados por meio do algoritmo *TeraGen* e consistem em dois arquivos, com 50 mil linhas cada. O formato do arquivo gerado pelo *TeraGen* é apresentado na Figura 5.1. Nela, pode-se observar que as chaves são compostas por valores aleatórios do conjunto  $\{‘ ‘, ‘..’, ‘~’\}$ , o identificador da linha (*id\_linha*) é um número inteiro e o valor consiste de sete sequências de dez caracteres de ‘A’ a ‘Z’. Cada linha desse arquivo corresponde a 100 Bytes, o que totaliza quase 10 MB de dados.

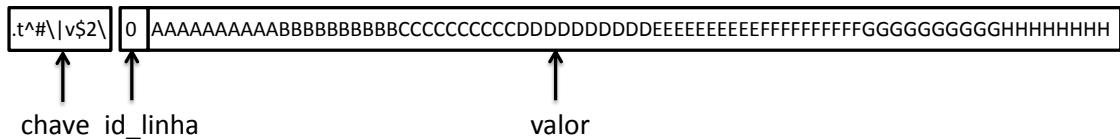


Figura 5.1: Formato do arquivo de dados gerado pelo *TeraGen*

Para o *benchmark Sort*, os dados para ordenação foram gerados pelo algoritmo *RandomWriter*, e consistem em 10 arquivos, em formato binário, de 1 GB cada, o que totaliza 10 GB de dados. Os arquivos são formados por chaves compostas por dois caracteres aleatórios. Esses caracteres podem ser combinações de números inteiros e/ou letras. São exemplos desse conjunto as chaves: ‘21’, ‘ba’, ‘b2’ e ‘2b’.

### 5.3.2 Algoritmo de Ordenação por Amostragem

Para gerar os dados para ordenação utilizados pelo algoritmo de Ordenação por Amostragem foi implementado um programa em Java para geração de chaves inteiras aleatórias. As chaves consistem em valores inteiros positivos entre 0 e  $2^{31} - 1$  (valor máximo de uma variável inteira). Foram geradas  $10^6$  (2 MB) a  $10^{10}$  (20 GB) chaves inteiras para ordenação, utilizando até dez cenários, isto é, dez diferentes conjuntos de dados aleatórios. As chaves

de cada cenário foram armazenadas em mais de um arquivo, em formato binário, conforme a quantidade gerada, para simular uma situação real em que é impossível ter os dados em uma só máquina. Dessa forma, quando  $10^8$  chaves foram geradas, elas foram distribuídas em 100 arquivos com  $10^6$  chaves cada, para cada cenário, conforme esquematizado na Figura 5.2. Da mesma forma, para uma quantidade de  $10^9$  chaves, foram gerados 1.000 arquivos com  $10^6$  chaves cada. Para uma quantidade de  $10^{10}$  chaves, foram gerados 10.000 arquivos com  $10^6$  chaves cada.

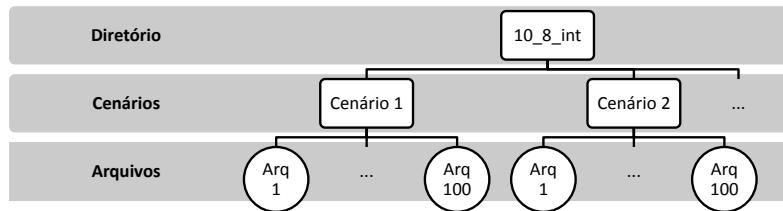


Figura 5.2: Esquema utilizado para geração de dados para ordenação pelo Algoritmo de Ordenação por Amostragem

#### 5.4 Parâmetros de Balanceamento

Em todos os testes do algoritmo de Ordenação por Amostragem foram utilizados os parâmetros de balanceamento apresentados na Tabela 5.1. O parâmetro frequência representa a probabilidade de uma amostra ser escolhida e vale 90%. O número máximo de amostras, cujo valor é 10.000, determina a quantidade máxima de amostras a serem escolhidas para estimar a distribuição de chaves do conjunto. O número máximo de partições ( $n_p$ ) corresponde à quantidade máxima de partições que podem ser utilizadas, seu valor é dado pela equação  $n_p = n_c \times n_{mag}$ , sendo  $n_c$  o número de cores e  $n_{mag}$  o número de máquinas. No caso dos experimentos feitos nesse trabalho,  $n_c$  é sempre igual a dois. Sendo assim, nos testes com duas máquinas,  $n_p$  é igual a quatro. Para três máquinas,  $n_p$  vale seis, e assim por diante.

<b>Frequência</b>	90%
<b>Nº máximo de amostras</b>	10.000
<b>Nº máximo de partições</b>	4 (2 máquinas)
	6 (3 máquinas)
	8 (4 máquinas)
	10 (5 máquinas)

Tabela 5.1: Parâmetros de balanceamento utilizados nos testes

# Capítulo 6

## Resultados

Neste capítulo são apresentados e analisados os resultados obtidos para os testes descritos no capítulo 5.

### 6.1 Testes Utilizando *Benchmarks* de Ordenação

Os resultados obtidos para o *benchmark* *TeraSort* são apresentados na Tabela 6.1. Observa-se que o algoritmo *TeraGen* realizou apenas tarefas Map (uma) e gastou 13 segundos para gerar os quase 10 MB de dados. O *TeraSort* realizou duas tarefas Map, uma tarefa Reduce e ordenou todos os quase 10 MB de dados em 21 segundos. O *TeraValidate* realizou uma tarefa Map, uma tarefa Reduce e gastou 21 segundos para validar os dados.

Algoritmo	Número de tarefas		Tempo (seg)
	Map	Reduce	
TeraGen	1	0	13
TeraSort	2	1	21
TeraValidate	1	1	21

Tabela 6.1: Resultados do *TeraSort* para 2 máquinas

A Tabela 6.2 apresenta os resultados obtidos para o *benchmark* *Sort*. Pode-se perceber que o *RandomWriter* executou 40 tarefas Map, nenhuma tarefa Reduce e gerou os dados em 5 minutos e 24 segundos (324 segundos). O *Sort* realizou 640 tarefas Map, 7 tarefas Reduce e ordenou os dados em 42 minutos e 1 segundo (2.521 segundos).

Algoritmo	Número de tarefas		Tempo (seg)
	Map	Reduce	
RandomWriter	40	0	324
Sort	640	7	2.521

Tabela 6.2: Resultados do *Sort* para 4 máquinas

A Tabela 6.3 apresenta uma comparação dos resultados obtidos pelos algoritmos *TeraSort* e *Sort*. Como pode-se perceber, o *Sort* ordenou 10 GB de dados utilizando quatro máquinas, uma quantidade de dados 1.000 vezes maior que a quantidade ordenada pelo *TeraSort* (10 MB em duas máquinas), em 2.521 segundos, tempo 100 vezes maior que o apresentado pelo *TeraSort* (21 segundos). Dessa forma, o *Sort* ordenou 0,9917 MB por

máquina por segundo, quantidade cerca de quatro vezes maior do que o *TeraSort*, que ordenou 0,2381 MB por máquina por segundo.

Algoritmo	Quantidade		Tempo (s)
	Dados (MB)	Máquinas	
TeraSort	10	2	21
Sort	10.000	4	2.521

Tabela 6.3: Comparação dos resultados do *TeraSort* e *Sort*

## 6.2 Testes de Verificação do Algoritmo

Observou-se que, na maioria dos testes de verificação, o algoritmo de Ordenação por Amostragem dividiu os dados em partições equilibradas (balanceadas), sejam eles valores inteiros aleatórios homogêneos (próximos) ou viciados (distantes) e para quantidades de dados pares e ímpares. Consequentemente, a variação dos tempos de ordenação foi relativamente pequena.

Os resultados obtidos também mostraram que é importante uma boa escolha dos parâmetros para o balanceamento, isto é, a frequência de escolha da amostra, o número de amostras a serem escolhidas e o número máximo de partições, pois estes parâmetros influenciam o tempo de ordenação. Além disso, os testes permitiram observar a aleatoriedade do algoritmo, uma vez que execuções com os mesmos parâmetros de balanceamento produziram resultados diferentes.

## 6.3 Testes de Estabilidade do Sistema

O gráfico da Figura 6.1 apresenta os tempos obtidos para cada execução do arquivo com  $10^6$  inteiros nos testes de estabilidade do sistema.

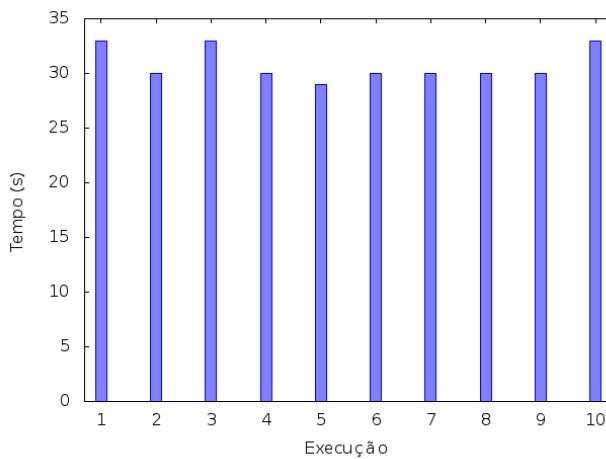


Figura 6.1: Gráfico dos tempos obtidos para 10 execuções de um arquivo com  $10^6$  inteiros aleatórios em 4 máquinas

A Tabela 6.4 apresenta os dados estatísticos dos tempos de ordenação obtidos nas dez

execuções do arquivo com  $10^6$  inteiros. Pode-se observar que o melhor tempo obtido foi 29 segundos, o pior tempo foi 33 segundos e o tempo médio das execuções foi de 31 segundos. A mediana, centro da distribuição dos dados, corresponde a 30 segundos e está próxima da média, o que demonstra que os tempos obtidos estão próximos um dos outros. Uma medida importante retratada nessa tabela é o coeficiente de variação, medida independente de grandeza da dispersão de uma amostra em relação à sua média, dada pela divisão do desvio padrão pela média, cujo valor é 0,0496. Esse valor indica que ocorreu uma pequena variação nos tempos, para o mesmo conjunto de elementos nas diferentes execuções, fato comprovado e melhor visualizado no gráfico da Figura 6.1. Tal resultado pode ser explicado pela seleção aleatória dos registros realizada pelo algoritmo para constituir as amostras responsáveis por estimar a distribuição de chaves no conjunto.

<b>Menor</b>	<b>Mediana</b>	<b>Média</b>	<b>Maior</b>	<b>Coef. de Variação</b>	<b>Q1</b>	<b>Q3</b>
29	30	31	33	0,0496	30	33

Tabela 6.4: Tempos estatísticos, em segundos, obtidos para 10 execuções de um arquivo com  $10^6$  inteiros aleatórios em 4 máquinas

Na Tabela 6.5 são apresentados os dados estatísticos dos números de elementos obtidos nas oito partições produzidas em cada uma das dez execuções do arquivo com  $10^6$  inteiros. Observa-se que o número de elementos nas partições está compreendido no intervalo de 114.548 a 132.502, seu valor médio corresponde a 125.052, enquanto o centro da distribuição dos dados é de 125.411. O coeficiente de variação igual a 0,0261 indica que ocorreu uma pequena variação no número de elementos nas partições. Sendo assim, pode-se concluir que as partições obtidas estão relativamente equilibradas.

<b>Menor</b>	<b>Mediana</b>	<b>Média</b>	<b>Maior</b>	<b>Coef. de Variação</b>	<b>Q1</b>	<b>Q3</b>
114.548	125.411	125.052	132.502	0,0261	122.492	127.270

Tabela 6.5: Número de elementos estatísticos das partições obtidas para 10 execuções de um arquivo com  $10^6$  inteiros aleatórios em 4 máquinas

O gráfico da Figura 6.2 apresenta a relação entre o balanceamento das partições obtidas e o tempo de ordenação em cada uma das dez execuções do arquivo com  $10^6$  inteiros. O percentual de proximidade entre os elementos da partição foi calculado pela razão entre o menor número de elementos obtido nas partições formadas e o maior número de elementos obtido. Quanto mais próximo o percentual está de um, maior é a proximidade entre os elementos, ou seja, mais balanceada está a partição. Pode-se observar que a proximidade entre os elementos das partições formadas é maior que 0,85 (85%), o que mostra que as partições estão balanceadas e que os tempos de ordenação estão próximos uns dos outros. Vale ressaltar que aparentemente são oito pontos no gráfico, mas na verdade, são dez pontos, pois os outros dois pontos estão sobrepostos. Esse fato reforça a variabilidade

baixa das partições.

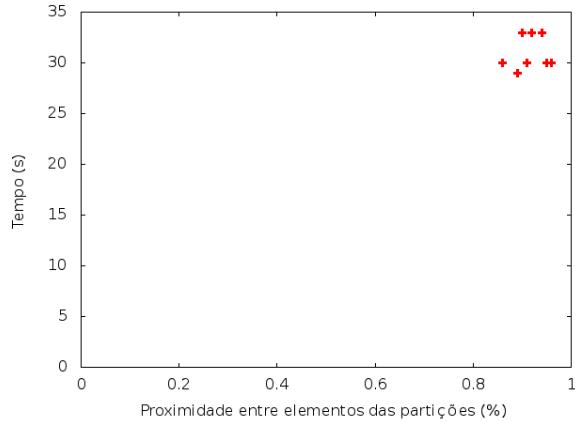


Figura 6.2: Gráfico da relação entre balanceamento e tempo de ordenação para 10 execuções de um arquivo com  $10^6$  inteiros aleatórios em 4 máquinas

#### 6.4 Testes Variando o Conjunto de Dados

Os tempos médios obtidos nas dez execuções de cada um dos dez conjuntos de dados são apresentados no gráfico da Figura 6.3.

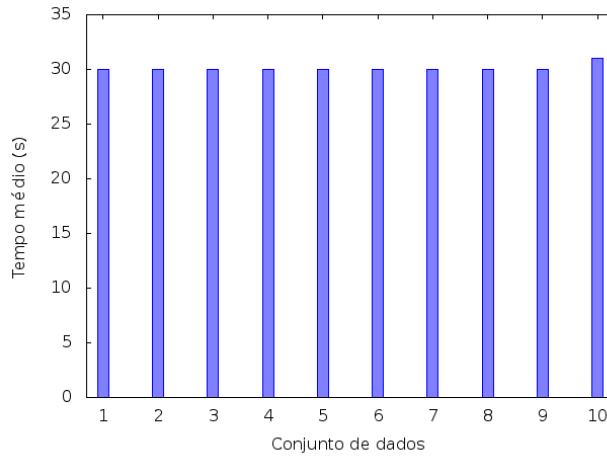


Figura 6.3: Gráfico dos tempos médios obtidos para 10 execuções de 10 conjuntos de  $10^6$  inteiros aleatórios em 4 máquinas

A Tabela 6.4 apresenta os dados estatísticos dos tempos médios de ordenação obtidos nas dez execuções dos diferentes conjuntos de dados de  $10^6$  inteiros. O melhor tempo obtido foi 30 segundos e o pior tempo foi 31 segundos. O tempo médio e a mediana foram iguais, o que demonstra simetria nos tempos obtidos. O coeficiente de variação, cujo valor é 0,0070, indica que ocorreu uma pequena variação nos tempos, o que mostra que o conjunto de entrada aleatório pouco interfere nos resultados. Esse fato pode ser comprovado e melhor

visualizado no gráfico da Figura 6.3.

<b>Menor</b>	<b>Mediana</b>	<b>Média</b>	<b>Maior</b>	<b>Coef. de Variação</b>	<b>Q1</b>	<b>Q3</b>
30	30	30	31	0,0070	30	30

Tabela 6.6: Tempos estatísticos, em segundos, obtidos para 10 execuções de 10 conjuntos de  $10^6$  inteiros aleatórios em 4 máquinas

Na Tabela 6.7 são apresentados os dados estatísticos dos números de elementos obtidos, para as oito partições produzidas, nas dez execuções dos dez conjuntos de dados de  $10^6$  inteiros. Pode-se observar que o número de elementos nas partições está compreendido no intervalo de 122.221 a 127.308, seu valor médio corresponde a 125.000, enquanto o centro da distribuição dos dados é de 124.862. O coeficiente de variação igual a 0,0081 indica que ocorreu uma pequena variação no número de elementos nas partições. Sendo assim, pode-se concluir que as partições obtidas estão relativamente equilibradas.

<b>Menor</b>	<b>Mediana</b>	<b>Média</b>	<b>Maior</b>	<b>Coef. de Variação</b>	<b>Q1</b>	<b>Q3</b>
122.221	124.862	125.000	127.308	0,0081	124.338	125.766

Tabela 6.7: Número de elementos estatísticos das partições obtidas para 10 execuções de 10 conjuntos de  $10^6$  inteiros aleatórios em 4 máquinas

## 6.5 Testes Variando a Quantidade de Dados

O gráfico da Figura 6.4 mostra os tempos médios obtidos em três execuções do algoritmo para conjuntos de dados de tamanhos  $10^6$  a  $10^{10}$ . Nesse gráfico, os eixos x e y estão em escala logarítmica para facilitar a visualização dos dados, visto que os tempos médios obtidos se encontram em várias ordens de grandeza, à medida que a quantidade de dados aumenta.

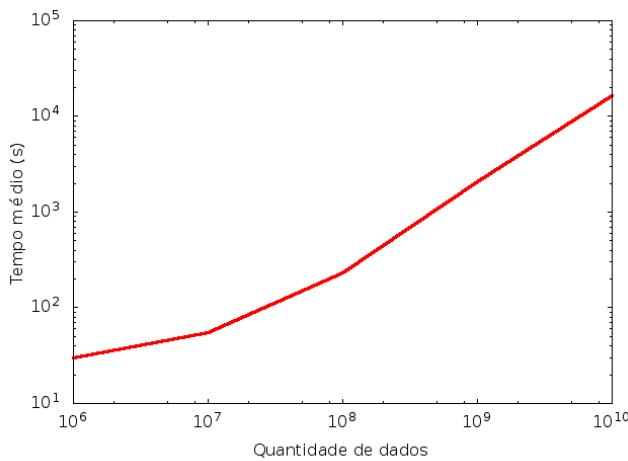


Figura 6.4: Gráfico dos tempos médios obtidos para  $10^6$  a  $10^{10}$  inteiros aleatórios em 4 máquinas

A Tabela 6.8 apresenta os dados estatísticos para os tempos obtidos em três execuções do algoritmo para as diferentes quantidades de dados. O tempo médio de ordenação para  $10^6$  chaves inteiras foi de aproximadamente 30 segundos. Para  $10^7$  chaves, o tempo médio correspondeu a 55 segundos. Para  $10^8$  chaves, o tempo médio foi em torno de 3 minutos. Já para  $10^9$  chaves, o tempo médio obtido foi em torno de 34 minutos. Para  $10^{10}$  chaves, o tempo médio aumentou consideravelmente, correspondendo a aproximadamente 4 horas. Observa-se que os valores do tempo médio e mediana obtidos para as diferentes quantidades de dados estão próximos. Além disso, pode-se perceber que o coeficiente de variação do tempo diminuiu à medida que a quantidade de dados aumentou.

Nº de dados	Menor	Mediana	Média	Maior	Coef. de Variação	Q1	Q3
$10^6$	29	30	30	30	0,0158	29	30
$10^7$	50	54	55	60	0,0754	50	60
$10^8$	230	231	232	235	0,0104	230	235
$10^9$	2.052	2.078	2.074	2.091	0,0078	2.052	2.091
$10^{10}$	16.321	16.367	16.429	16.599	0,0074	16.321	16.599

Tabela 6.8: Tempos estatísticos, em segundos, obtidos para  $10^6$  a  $10^{10}$  inteiros aleatórios em 4 máquinas

Com a finalidade de dimensionar o *overhead* do algoritmo, isto é, a sobrecarga ou custo adicional de comunicação entre os processadores mestre e escravos, assim como sua escalabilidade em relação ao número de dados, foi calculado o tempo médio relativo de ordenação para cada  $10^6$  dados. Dessa forma, sendo um Megadado equivalente a  $10^6$  dados, o tempo médio resultante para ordenar dez Megadados ( $10^7$  dados) é 5,5 segundos por megadado. O tempo médio para ordenar cem Megadados ( $10^8$  dados) vale 2,32 segundos por megadado, e assim por diante. O gráfico da Figura 6.5 representa esses resultados. Pode-se observar que, à medida que a quantidade de Megadados para ordenação aumenta, o tempo médio para ordenação diminui. Essa diminuição no *overhead* pode ser explicada por uma melhor distribuição da carga de trabalho entre os oito processadores participantes. Além disso, observa-se uma boa escalabilidade do algoritmo em relação ao número de dados, pois seu desempenho, medido em tempo, melhorou à medida que a quantidade de dados aumentou.

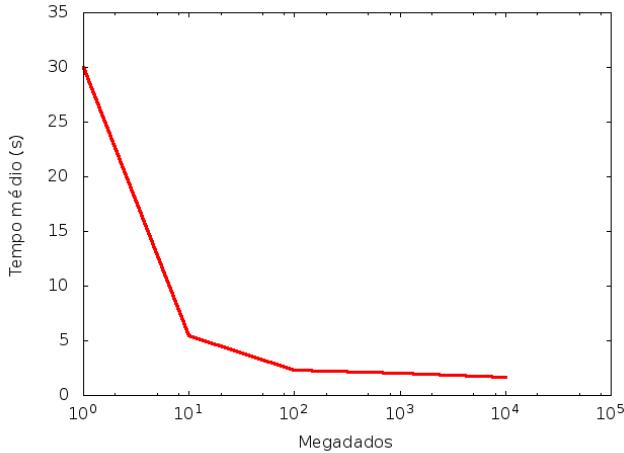
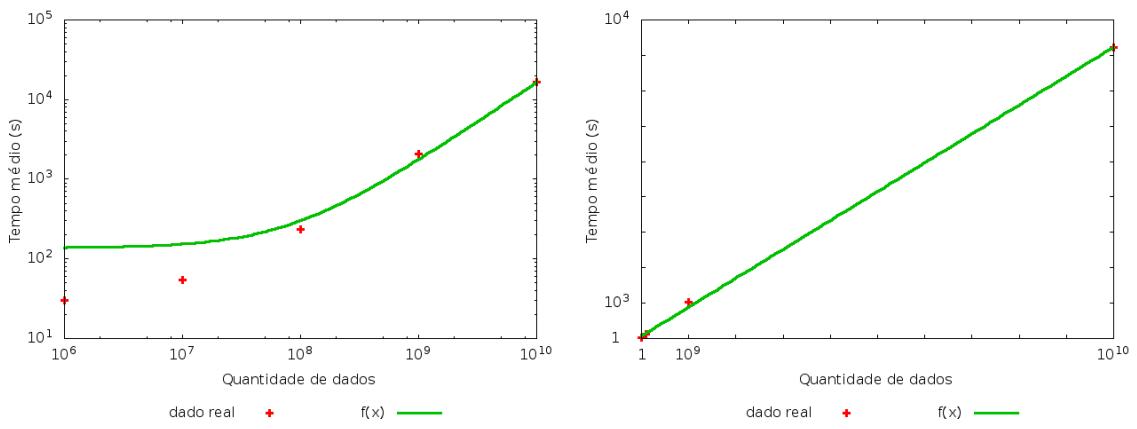


Figura 6.5: Gráfico dos tempos médios obtidos para cada  $10^6$  dados aleatórios em 4 máquinas

Uma interpolação do gráfico da Figura 6.4 foi feita com o objetivo de estimar a complexidade do algoritmo para os testes apresentados nesta seção. O gráfico obtido, apresentado na Figura 6.6a com os eixos em escala logarítmica e na Figura 6.6b com os eixos em escala linear, exibe o dado real e a aproximação linear ( $f(x)$ ) encontrada para o mesmo. A equação da aproximação linear obtida é dada por  $f(x) = 1,632 \times 10^{-6}x + 137$  e sua correlação ( $r$ ) entre o dado real e o estimado por ela corresponde a 0,9997. O fato da correlação obtida ser positiva e bastante próxima de um, indica que as duas variáveis se movem juntas, e a correlação entre elas é forte, ou seja, a função obtida é uma boa aproximação para o conjunto. A complexidade linear ( $O(n)$ ) obtida é esperada para o algoritmo, pois a ordenação externa pressupõe leituras e escritas dos itens do arquivo em memória auxiliar, assim como a divisão dos dados entre os processadores participantes do processo.



(a) Aproximação linear com eixos em escala logarítmica (b) Aproximação linear com eixos em escala linear

Figura 6.6: Gráfico de interpolação para o gráfico da Figura 6.4

## 6.6 Testes Variando a Quantidade de Máquinas

O gráfico da Figura 6.7 mostra como os tempos médios obtidos nas três execuções de um arquivo com  $10^8$  inteiros aleatórios diminuíram com o aumento do número de máquinas. Essa diminuição pode ser explicada pela maior utilização da memória interna e, consequentemente, menor troca de dados entre a memória interna e externa.

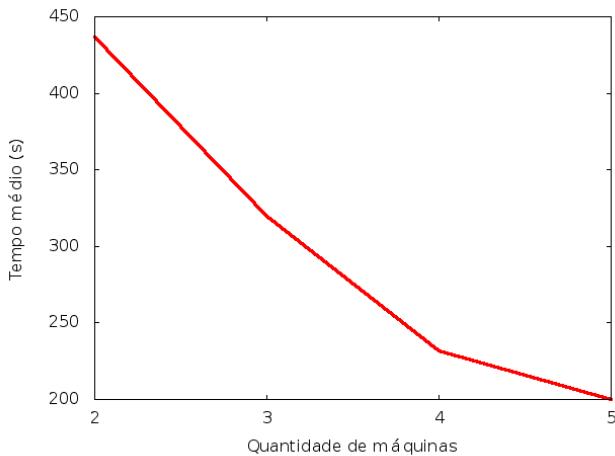


Figura 6.7: Gráfico dos tempos médios obtidos para três execuções de  $10^8$  inteiros aleatórios em 2 a 5 máquinas

A Tabela 6.9 apresenta os dados estatísticos dos tempos obtidos em três execuções do algoritmo para  $10^8$  inteiros em conjuntos de duas a cinco máquinas. O tempo médio obtido para duas máquinas foi de aproximadamente 7 minutos. Para três máquinas, o tempo médio foi em torno de 5 minutos. Para quatro máquinas, o tempo médio foi cerca de 4 minutos, em torno de 47% menor que o tempo médio obtido com duas máquinas. Para cinco máquinas, o tempo médio foi aproximadamente 3 minutos, cerca de 54% menor que o tempo médio obtido com duas máquinas.

Nº de máquinas	Menor	Mediana	Média	Maior	Coef. de Variação	Q1	Q3
2	419	426	437	467	0,0489	419	467
3	297	299	320	363	0,0961	297	363
4	230	231	232	235	0,0104	230	235
5	194	194	200	213	0,0453	194	213

Tabela 6.9: Tempos estatísticos, em segundos, para  $10^8$  inteiros aleatórios em diferentes quantidades de máquinas

Com o intuito de estimar a escalabilidade do algoritmo em relação ao número de máquinas, foram calculadas as medidas de desempenho denominadas *speedup* e eficiência, descritas na seção 3.2. As fórmulas foram adaptadas para verificar a melhora de desempenho e eficiência obtidas partindo-se de duas máquinas (quatro processadores), ao invés de uma máquina (um processador, execução sequencial). Por exemplo, o *speedup* ideal foi

calculado para seis processadores, tomando-se como base quatro processadores pela razão 4/6. O *speedup* respectivo (real) foi calculado pela razão entre o tempo médio de execução com quatro processadores e o tempo médio de execução com seis processadores, e assim por diante. A eficiência foi calculada pela divisão do *speedup* real pelo *speedup* ideal.

A Tabela 6.10 apresenta o ganho de desempenho (*speedup*) obtido com o aumento do número de processadores, bem como a melhora desejada (ideal). O gráfico da Figura 6.8 representa graficamente os resultados obtidos e permite observar que, em geral, o *speedup* obtido foi próximo ao valor ideal, porém menor que o valor desejado (*speedup* sublinear). O aumento de quatro para oito processadores, por exemplo, tornou a ordenação cerca de 1,88 vezes mais rápida, enquanto o ideal era tornar a ordenação duas vezes mais rápida. Esse resultado é esperado, pois a obtenção do *speedup* ideal é considerada uma rara ocorrência, uma vez que a maioria das soluções paralelas introduzem alguma sobrecarga produto da distribuição de carga (balanceamento) e comunicação entre processos, o que pode ser observado no algoritmo de Ordenação por Amostragem.

Nº processadores	Tempo médio (s)	Speedup real	Speedup ideal
4	437	1,00	1,00
6	320	1,37	1,50
8	232	1,88	2,00
10	200	2,19	2,50

Tabela 6.10: *Speedup* para  $10^8$  inteiros aleatórios em 4 a 10 processadores

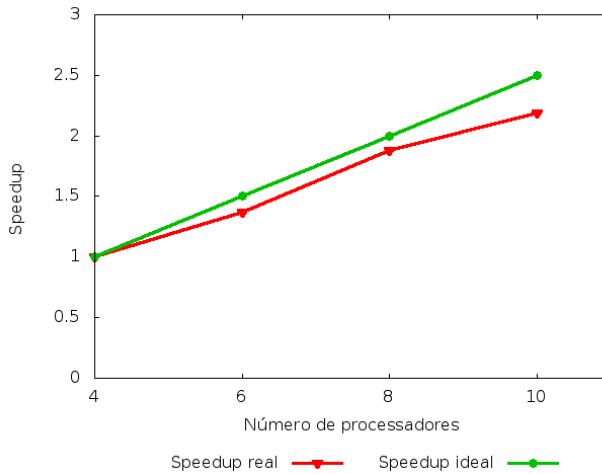


Figura 6.8: Gráfico da relação entre o número de processadores e o *speedup* para  $10^8$  inteiros aleatórios

A Tabela 6.11 apresenta o percentual de utilização dos processadores (eficiência) resultante do aumento de processadores. O gráfico da Figura 6.9 representa os resultados obtidos (eficiência real) em relação à eficiência ideal (100%) e permite observar que, em geral, a eficiência foi próxima ao valor ideal, porém menor que o valor desejado (eficiência máxima). O aumento de quatro para seis processadores, por exemplo, resultou em 91% de

eficiência. Já o aumento para oito processadores proporcionou uma eficiência de 94%, o que indica uma forte utilização dos processadores. O aumento para dez processadores gerou 87% de eficiência, isso indica diminuição da eficiência decorrente provavelmente da melhor distribuição da carga de trabalho, que faz com que cada processador seja responsável por executar uma menor quantidade de trabalho. Apesar da eficiência não ter sido mantida constante com o aumento do número de processadores, o algoritmo pode ser considerado escalável, uma vez que os valores de eficiência obtidos foram próximos uns dos outros.

Nº processadores	Eficiência (%)
4	100,00
6	91,04
8	94,18
10	87,40

Tabela 6.11: Eficiência para  $10^8$  inteiros aleatórios em 4 a 10 processadores

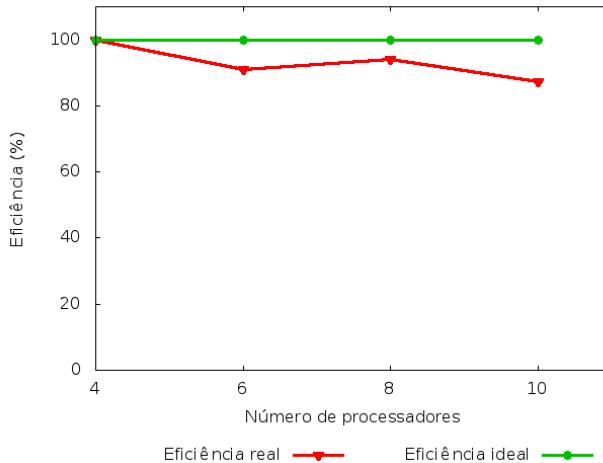


Figura 6.9: Gráfico da relação entre o número de processadores e a eficiência para  $10^8$  inteiros aleatórios

## 6.7 Considerações Finais

Os resultados descritos neste capítulo permitiram obter as seguintes conclusões parciais em relação ao algoritmo de Ordenação por Amostragem:

- **Formação de partições balanceadas:** baixa variabilidade no número de elementos das partições, uma vez que o coeficiente de variação obtido foi 0,0261. Além disso, a proximidade entre os elementos das partições é maior que 85%.
- **Baixa variação dos tempos de ordenação:** pequena variação do tempo (0,0496) para dez execuções do mesmo conjunto de dados, o que demonstra estabilidade do sistema. Para alteração do conjunto de dados, a variação do tempo foi de apenas 0,0070, o que indica pouca interferência do conjunto aleatório de dados gerado no desempenho do algoritmo.

- **Influência dos parâmetros para balanceamento:** o tempo de ordenação depende de uma boa escolha dos parâmetros para balanceamento e execuções com os mesmos parâmetros podem produzir resultados diferentes.
- **Escalabilidade em relação ao número de dados:** o desempenho, medido em tempo, melhorou à medida que a quantidade de Megadados (cada  $10^6$  dados) para ordenação aumentou.
- **Complexidade linear em relação ao tempo de execução:** o tempo de ordenação aumenta linearmente com a quantidade de dados a ser ordenada, uma vez que o algoritmo realiza operações de leituras e escritas dos dados em memória auxiliar e divisão dos dados entre os processadores.
- **Speedup sublinear:** o ganho de desempenho (*speedup*) com o aumento do número de processadores (4 a 10) foi próximo, porém inferior ao valor ideal devido à sobrecarga introduzida pelo paralelismo para balanceamento de carga e comunicação entre processos.
- **Eficiência sublinear:** o percentual de utilização dos processadores (eficiência) obtido com o aumento do número de processadores (4 a 10) foi próximo, porém menor que o valor ideal.
- **Escalabilidade em relação ao número de máquinas:** os valores de eficiência obtidos foram próximos uns dos outros, apesar de não se manterem constantes com o aumento do número de processadores.

## Capítulo 7

# Conclusões e Perspectivas

### 7.1 Conclusões

O problema investigado por este trabalho consistiu na avaliação da escalabilidade da ordenação paralela em relação ao número de dados a serem ordenados e ao número de máquinas utilizadas, com os objetivos de analisar as potencialidades e limitações da computação paralela em máquinas *multicore*, aplicada ao paradigma da ordenação no ambiente Hadoop.

A fim de alcançar estes objetivos, foram estudados os princípios da computação paralela e de versões paralelas de algoritmos de ordenação e, então, implementado o algoritmo de Ordenação por Amostragem em Java, de acordo com o modelo MapReduce, no ambiente Hadoop. Esse algoritmo é diferente dos demais por utilizar uma estratégia de amostragem, que busca estimar a distribuição de chaves no conjunto e, assim criar partições balanceadas, a partir da análise de amostras, ou seja, de um subconjunto de dados, ao invés do conjunto inteiro.

Os resultados apresentados nos testes realizados mostraram que o algoritmo dividiu os dados em partições equilibradas (balanceadas), sejam eles valores inteiros aleatórios homogêneos (próximos) ou viciados (distantes) e para quantidades de dados pares e ímpares. O balanceamento das partições é um fator muito importante, pois distribui a carga de trabalho entre os processadores, de modo a evitar que um processador esteja ocioso enquanto outro está sobrecarregado e, assim, contribui para melhor desempenho, ou seja, para um menor tempo de ordenação.

Além disso, foi possível perceber que conjuntos de entrada aleatórios formados por distribuições de dados uniformes pouco interferem nos tempos de ordenação, o que mostra que o algoritmo não está restrito a apenas alguns conjuntos de dados de uma distribuição.

Em relação à quantidade de dados, o tempo para ordenar  $10^6$  chaves inteiras (30 segundos, tamanho total de 2MB) foi bastante diferente do obtido para  $10^{10}$  chaves (4 horas, tamanho total de 20 GB). Esse resultado evidencia o fato de que à medida que a quantidade de dados aumenta, a computação paralela pode tornar as operações mais eficientes e, até mesmo, viáveis. O cálculo do tempo médio relativo de ordenação para cada  $10^6$  dados (Megadado) mostrou que o desempenho do algoritmo aumentou à medida que à complexi-

dade do problema aumentou, o que evidencia boa escalabilidade do algoritmo em relação ao número de dados e baixo *overhead* para comunicação entre os processadores mestre e escravos.

O aumento do número de máquinas para ordenação de  $10^8$  chaves (corresponde a um tamanho de 200 MB) acarretou uma diminuição de no máximo 54% no tempo. É possível que essa redução seja maior para uma quantidade maior de dados, o que indica que a computação paralela, em relação ao número de máquinas, pode reduzir significativamente o tempo de processamento apenas para quantidades massivas de dados. Os resultados obtidos para o *speedup* e eficiência foram em geral próximos e menores que os respectivos valores ideais, uma vez que o paralelismo introduz alguma sobrecarga para balanceamento de carga e comunicação entre processos. O aumento de quatro para oito processadores, por exemplo, tornou a ordenação cerca de 1,88 vezes mais rápida, enquanto o ideal era tornar a ordenação duas vezes mais rápida. Para essa mesma configuração, foi obtida 94% de eficiência, ou seja, uma forte utilização dos processadores. Embora a eficiência não tenha sido mantida constante com o aumento do número de processadores, pode-se verificar a escalabilidade do algoritmo em relação ao número de máquinas, uma vez que os valores de eficiência obtidos foram próximos uns dos outros.

Por outro lado, o algoritmo é dependente de uma boa escolha dos parâmetros para o balanceamento, isto é, a frequência de escolha da amostra, o número de amostras a serem escolhidas e o número máximo de partições, pois estes parâmetros influenciam no tempo de ordenação. O algoritmo também apresenta certa aleatoriedade, uma vez que execuções com os mesmos parâmetros de balanceamento produziram resultados diferentes.

A quantidade cada vez maior de dados manipulados pelas aplicações torna os algoritmos de ordenação sequenciais ineficientes para ordenar dados em tempo aceitável. Neste contexto, a ordenação paralela surge como uma solução e uma necessidade, visto que o aumento da velocidade de processamento para obter maior desempenho atingiu seu limite e os processadores *multicore* são uma realidade atualmente. A programação paralela é a única opção para uso eficiente dos recursos oferecidos pelos processadores *multicore*, cujos núcleos de processamento tendem a continuar aumentando. Portanto, é fundamental sua aprendizagem.

Devido ao grande número de algoritmos de ordenação paralela e a variedade de arquiteturas paralelas, são necessários estudos para avaliar e selecionar algoritmos adequados para multiprocessadores.

Entretanto, a programação paralela introduz novos desafios: os programadores precisam pensar de uma maneira difícil e diferente da que estão acostumados. É preciso preocupar-se com múltiplos fluxos executando ao mesmo tempo, em como coordená-los e com todo um novo conjunto de erros e problemas de desempenho que não têm equivalente na programação serial, tais como disputa de dados e *deadlocks* [Breshears 2009].

O Hadoop é uma plataforma de computação distribuída que facilita a implementação

e execução de aplicações no paradigma de programação paralelo, que processam grandes quantidades de dados, por apresentar características, como as seguintes:

- **Hadoop Distributed FileSystem (HDFS):** sistema de arquivo distribuído, que se encarrega de distribuir os dados entre as máquinas de maneira transparente, como se os dados estivessem armazenados localmente;
- **Implementação do paradigma de programação MapReduce:** as funções Map e Reduce estruturam a implementação das aplicações e são responsáveis por agendar, monitorar e reexecutar tarefas, em caso de falha;
- **Escalabilidade:** permite armazenar e processar PetaBytes, de forma confiável. O armazenamento e processamento do HDFS é feito em cada nó do *cluster*, ao contrário dos demais sistemas de arquivos;
- **Economia:** distribui os dados e os processa entre *clusters* de computadores comumente disponíveis. Esses *clusters* podem ter milhares de nós (máquinas);
- **Eficiência:** ao distribuir os dados, o Hadoop pode processá-los em paralelo por meio dos nós, onde os dados estão localizados, de forma rápida;
- **Confiabilidade:** múltiplas cópias dos dados são automaticamente mantidas e tarefas remanejadas, em caso de falha.

O modelo de programação MapReduce presente no Hadoop, ao contrário de modelos como o MPI (*Message Passing Interface*), opera em alto nível, ou seja, o programador pensa em termos de funções estruturadas em pares (chave, valor) e o fluxo de dados está implícito. Dessa forma, o programador não precisa lidar explicitamente com mecanismos de fluxo de dados de baixo nível, tais como *sockets*. Coordenar os processos em uma computação distribuída de larga escala é um desafio. O aspecto considerado mais difícil é o tratamento de falhas. O MapReduce, ao contrário do MPI, poupa o programador de ter de gerenciar explicitamente falhas e recuperações das mesmas, uma vez que detecta automaticamente falhas de tarefas Map e Reduce e reagenda suas execuções em máquinas cujo funcionamento não foi comprometido.

Um grande conjunto de algoritmos podem ser expressos em MapReduce como a análise de imagem, problemas baseados em grafos e algoritmos de aprendizado de máquina. Apesar de não ser capaz de resolver todos os tipos de problemas, o MapReduce é uma ferramenta de processamento de dados em geral que podem ser estruturados em pares (chave, valor) de modos específicos.

Devido à essa grande variedade de recursos, o Hadoop está sendo cada vez mais utilizado para processamento e análise de quantidades massivas de dados em redes sociais e projetos de grandes proporções como o sequenciamento do DNA (Projeto Genoma). Nem todo usuário do Hadoop demanda uma escala massiva de dados ao nível de empresas como

Google e Facebook, mas empresas com razoável volume de informações não estruturadas, como bancos, varejo e empresas aéreas podem utilizar o Hadoop como uma boa ferramenta para tratamento analítico dos seus dados.

Este trabalho representou para mim uma oportunidade única para aquisição e compreensão de conceitos importantes referentes à arquitetura, programação e ordenação paralela, uma vez que tive apenas uma visão superficial e breve contato com a área durante o curso, que considero um diferencial da Engenharia de Computação em relação aos demais cursos de computação. Por meio dele, percebi a importância e a necessidade da programação paralela para aproveitar todos os recursos dos computadores atuais e obter maior desempenho. Além disso, conheci o funcionamento do ambiente Hadoop e sua diversidade de recursos para facilitar a implementação do paralelismo, no nível de tarefas, em um sistema distribuído. Também aprendi como o modelo MapReduce pode ser utilizado para estruturar programas de ordenação paralela e diminuir a complexidade de desenvolver programas paralelos, principalmente por gerenciar automaticamente o tratamento de falhas, questão fundamental de confiabilidade de um sistema. Com a disponibilidade de processadores *multicore* e a quantidade cada vez maior de dados manipulados pelas aplicações, acredito que existe ampla possibilidade de futuras aplicações dos conhecimentos adquiridos e conceitos explorados neste trabalho.

Os resultados da avaliação do algoritmo de Ordenação por Amostragem, implementado em MapReduce no ambiente Hadoop, em termos do número de dados e de máquinas utilizadas para ordenação são as principais contribuições para a comunidade acadêmica, isto é, a escalabilidade da solução, uma das principais características determinantes de desempenho em sistemas distribuídos.

## 7.2 Trabalhos Futuros

Como perspectivas de continuação deste trabalho, destacam-se a implementação, análise e comparação de versões paralelas de outros algoritmos de ordenação, como o *QuickSort*, *BubbleSort* e *RadixSort*, que foram discutidas no trabalho; a execução dos algoritmos em um *cluster* formado por mais máquinas e, assim, a utilização de variabilidade e quantidade massiva de dados nos testes, como dados gerados por outras distribuições diferentes de uniforme e dados obtidos de sistemas reais; a utilização de ferramenta de *profile* para a análise de informações das máquinas, tais como consumo de memória e desempenho de CPU, para cada tarefa executada.

Além desses trabalhos, também seria interessante explorar as limitações do ambiente Hadoop e detalhar os problemas que podem ser resolvidos, de forma eficiente, pelo modelo MapReduce.

## Referências Bibliográficas

- [Adve et al. 2008]ADVE, S. et al. Parallel Computing Research at Illinois: The UPCRC Agenda. University of Illinois, Urbana-Champaign, Illinois, USA, November 2008.
- [Aho e Hopcroft 1974]AHO, A.; HOPCROFT, J. *The Design and Analysis of Computer Algorithms*. 1st. ed. Boston, MA, USA: Addison-Wesley, 1974.
- [Almasi e Gottlieb 1989]ALMASI, G.; GOTTLIEB, A. *Highly Parallel Computing*. Redwood City, CA: Benjamin-Cummings, 1989.
- [Almasi e Gottlieb 1994]ALMASI, G.; GOTTLIEB, A. *Highly Parallel Computing*. 2nd. ed. Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc., 1994.
- [Amato et al. 1998]AMATO, N. et al. *A Comparison of Parallel Sorting Algorithms on Different Architectures*. College Station, TX, USA, 1998.
- [Amdahl 1967]AMDAHL, G. Validity of the single processor approach to achieving large scale computing capabilities. In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. New York, USA: ACM, 1967. p. 483–485.
- [Asanovic et al. 2006]ASANOVIC, K. et al. The Landscape of Parallel Computing Research: A View from Berkeley. University of California, Berkeley, CA, USA, December 2006.
- [Asanovic et al. 2008]ASANOVIC, K. et al. The Parallel Computing Laboratory at U.C. Berkeley: A Research Agenda Based on the Berkeley View. University of California, Berkeley, CA, USA, March 2008.
- [Asanovic et al. 2009]ASANOVIC, K. et al. A View of the Parallel Computing Landscape. *Commun. ACM*, ACM, New York, NY, USA, v. 52, p. 56–67, October 2009.
- [Batcher 1968]BATCHER, K. Sorting Networks and their Applications. In: *Proceedings of the April 30-May 2, 1968, Spring Joint Computer Conference*. New York, NY, USA: ACM, 1968. (AFIPS '68 (Spring)), p. 307–314.
- [Blelloch 1990]BLELLOCH, G. *Vector Models for Data-Parallel Computing*. Cambridge, MA, USA: MIT Press, 1990.

- [Blelloch, G. et al 1991] Blelloch, G. et al. A Comparison of Sorting Algorithms for the Connection Machine CM-2. In: *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures*. New York, NY, USA: ACM, 1991. (SPAA '91), p. 3–16.
- [Bondi 2000] BONDI, A. Characteristics of Scalability and their Impact on Performance. In: *Proceedings of the 2nd International Workshop on Software and Performance*. New York, NY, USA: ACM, 2000. (WOSP '00), p. 195–203.
- [Breshears 2009] BRESHEARS, C. *The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications*. 1st. ed. Sebastopol, CA, USA: O'Reilly Media, Inc., 2009.
- [Cassel et al. 2008] CASSEL, L. et al. *Computer Science Curriculum 2008: An Interim Revision of CS 2001*. 2008. <http://www.acm.org/education/curricula/ComputerScience2008.pdf>. Acessado em Maio de 2011.
- [Cole 1988] COLE, R. Parallel Merge Sort. *SIAM Journal on Computing*, v. 17, p. 770–785, 1988.
- [Culler e Singh 1999] CULLER, D.; SINGH, J. *Parallel Computer Architecture: a Hardware/Software Approach*. 1st. ed. San Francisco, California: Morgan Kaufmann Publishers, Inc, 1999. 1025 p.
- [Dean e Ghemawat 2008] DEAN, J.; GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, New York, USA, v. 51, n. 1, p. 107–113, 2008.
- [Flynn 1966] FLYNN, M. Very High-Speed Computing Systems. *Proceedings of The IEEE*, v. 54, p. 1901–1909, 1966.
- [Gray 1998] GRAY, J. *Sort Benchmark Homepage*. 1998. <http://sortbenchmark.org/>. Acessado em Junho de 2011.
- [Hadoop 2010] HADOOP. *Welcome to Apache Hadoop!* 2010. <http://hadoop.apache.org>. Acessado em Maio de 2011.
- [Held 2006] HELD, J. From a Few Cores to Many: A Tera-scale Computing Research Overview. *Intel Technology*, Intel White Paper, 2006.
- [Hennessy e Patterson 2007] HENNESSY, J.; PATTERSON, D. *Computer Architecture - A Quantitative Approach*. 4th. ed. San Francisco, California: Morgan Kaufmann, 2007.
- [Huang e Chow 1983] HUANG, J.; CHOW, Y. Parallel Sorting and Data Partitioning by Sampling. In: *Proceedings of the seventh International Computer Software and Applications Conference*. Washington, DC, USA: IEEE Computer Society, 1983. p. 627–631.

- [Kale e Solomonik 2010] KALE, V.; SOLOMONIK, E. Parallel Sorting Pattern. In: *Proceedings of the 2010 Workshop on Parallel Programming Patterns*. New York, NY, USA: ACM, 2010. (ParaPLoP '10).
- [Knuth 1973] KNUTH, D. *The Art of Computer Programming, Vol 3: Sorting and Searching*. 2nd. ed. Reading, MA: Addison-Wesley, 1973.
- [Kogge 2005] KOGGE, P. An Exploration of the Technology Space for Multi-Core Memory/Logic Chips for Highly Scalable Parallel Systems. In: *Proceedings of the Innovative Architecture on Future Generation High-Performance Processors and Systems*. Washington, DC, USA: IEEE Computer Society, 2005. p. 55–64.
- [Leighton 1992] LEIGHTON, F. *Introduction to Parallel Algorithms and Architectures - Arrays, Trees, Hypercubes*. San Francisco, CA, USA: Morgan Kaufmann, 1992.
- [Leighton 1985] LEIGHTON, T. Tight Bounds on the Complexity of Parallel Sorting. *IEEE Trans. Comput.*, IEEE Computer Society, Washington, DC, USA, v. 34, p. 344–354, April 1985.
- [Leiserson e Mirman 2008] LEISERSON, C.; MIRMAN, I. *How to Survive the Multicore Software Revolution (or at Least Survive the Hype)*. Cambridge, MA: Cilk Arts, 2008.
- [Li e Sevcik 1994] LI, H.; SEVCIK, K. Parallel Sorting by Over Partitioning. In: *Proceedings of the sixth annual ACM Symposium on Parallel Algorithms and Architectures*. New York, NY, USA: ACM, 1994. (SPAA '94), p. 46–56.
- [Migliacci 2009] MIGLIACCI, P. *Software livre Hadoop ganha espaço em grandes sites*. 2009. <http://tecnologia.terra.com.br/interna/0,,OI3642666-EI4801,00-Software+livre+Hadoop+ganha+espaco+em+grandes+sites.html>. Acessado em Maio de 2011.
- [Moore 1965] MOORE, G. Cramming more Components onto Integrated Circuits. *Electronics*, v. 38, n. 8, p. 114–117, April 1965.
- [O'Malley e Murthy 2009] O'MALLEY, O.; MURTHY, A. Winning a 60 second Dash with a Yellow Elephant. Yahoo!, Sunnyvale, CA, USA, 2009.
- [Pasin e Kreutz 2003] PASIN, M.; KREUTZ, D. Arquitetura e Administração de Aglomerados. In: *Anais da 3a. Escola Regional de Alto Desempenho*. Santa Maria, RS: Santa Maria, SBC, 2003. p. 3–34.
- [Quinn 1994] QUINN, M. *Parallel Computing: Theory and Practice*. 2nd. ed. New York, NY, USA: McGraw-Hill, Inc., 1994.

- [Rajasekaran e Reif 1989] RAJASEKARAN, S.; REIF, J. Optimal and Sublogarithmic Time Randomized Parallel Sorting Algorithms. *SIAM J. Comput.*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, v. 18, p. 594–607, June 1989.
- [Ranger et al. 2007] RANGER, C. et al. Evaluating MapReduce for Multicore and Multiprocessor Systems. In: *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2007. p. 13–24.
- [Rauber e Rünger 2010] RAUBER, T.; RÜNGER, G. *Parallel Programming for Multicore and Cluster Systems*. Berlin, Heidelberg, New York: Springer Verlag, 2010.
- [Reif e Valiant 1987] REIF, J.; VALIANT, L. A Logarithmic Time Sort for Linear Size Networks. *J. ACM*, ACM, New York, NY, USA, v. 34, p. 60–76, January 1987.
- [Robat 2007] ROBAT, C. *Introduction to Supercomputers*. 2007. <http://www.thocp.net/hardware/supercomputers.htm>. Acessado em Junho de 2011.
- [Shi e Schaeffer 1992] SHI, H.; SCHAEFFER, J. Parallel Sorting by Regular Sampling. *J. Parallel Distrib. Comput.*, Academic Press, Inc., Orlando, FL, USA, v. 14, p. 361–372, April 1992.
- [Shustek 2009] SHUSTEK, L. Interview: An interview with C.A.R. Hoare. *Commun. ACM*, ACM, New York, NY, USA, v. 52, p. 38–41, March 2009.
- [Silberschatz e Galvin 2004] SILBERSCHATZ, A.; GALVIN, P. *Sistemas Operacionais com Java*. 6th. ed. São Paulo: Prentice Hall, 2004.
- [Sutter e Larus 2005] SUTTER, H.; LARUS, J. Software and the Concurrency Revolution. *Queue*, ACM, New York, NY, USA, v. 3, p. 54–62, September 2005.
- [Tally 2007] TALLY, S. “*Not So Fast, Supercomputers*”, say Software Programmers. 2007. <http://news.uns.purdue.edu/x/2007a/070522SaiedParallel.html>. Acessado em Maio de 2011.
- [Tanembaum 2007] TANEMBAUM, A. *Organização Estruturada de Computadores*. 5a. ed. São Paulo: Pearson Education, 2007.
- [Venner 2009] VENNER, J. *Pro Hadoop*. Berkeley, CA, USA: Apress, 2009.
- [White 2009] WHITE, T. *Hadoop: The Definitive Guide*. 1st. ed. Sebastopol, CA, USA: O'Reilly Media, 2009.
- [Ziviani 2007] ZIVIANI, N. *Projeto de Algoritmos com implementações em Java e C++*. São Paulo, Brasil: Thomson Learning, 2007.