Alisson Gomes Cerqueira

Geração de Arquivo Invertido Utilizando Programação Paralela — MPI

Monografia de Graduação apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras como parte das exigências da disciplina Projeto Orientado para obtenção do título de Bacharel em Ciência da Computação.

Orientadora
Profa. Olinda Nogueira Paes Cardoso

Lavras
Minas Gerais - Brasil
2002

Alisson Gomes Cerqueira

Geração de Arquivo Invertido Utilizando Programação Paralela — MPI

Monografia de Graduação apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras como parte das exigências da disciplina Projeto Orientado para obtenção do título de Bacharel em Ciência da Computação.

Aprovada em *09 de Agosto de 2002*

____

Profa. Olinda Nogueira Paes Cardoso

____

Prof. Renato de Souza Gomes

____

Profa. Olinda Nogueira Paes Cardoso
(Orientadora)

Lavras
Minas Gerais - Brasil

Dedico este trabalho
Aos Professores Jones e Olinda, pela iniciativa da criação do projeto e pela orientação.
E a todos os colegas de turma pelos quatro anos.

Agradecimentos

A Mainha por sempre estar presente e
por sempre ter se dedicado aos filhos.
A minha irmã Mira pelo apoio de sempre.
A todos os colegas que fizeram parte do apto 210.
A Rubem Alves, Nietzsche e adeus.

Resumo

É proposto, nesse trabalho, o estudo da área de Recuperação de Informação nos modelos de indexação de coleções de documentos num ambiente paralelo. Um estudo dos principais algoritmos para geração de índice em paralelo e apresentação de um algoritmo em paralelo é realizado. O algoritmo apresentado foi implementado parcialmente, sendo que as principais operações realizadas sobre uma coleção foram concluídas com êxito. Os principais conceitos da computação paralela, MPI e de Recuperação de Informação são introduzidos por serem importantes para o entendimento do assunto.

Sumário

1 Introdução 1

2 Computação Paralela 3

2.1 Modelos de Arquitetura Paralela e Distribuída 3
2.2 Principais Conceitos 4
2.3 Tipos de Paralelismo 8
2.4 Escalabilidade e Concorrência 8
2.5 Speedup 9
2.6 Lei de Amdahl 10
2.7 Modelo PRAM 12
2.8 Ambiente Paralelo em Sistemas Distribuídos 14
2.9 MPI 15

3 Recuperação de Informação 19

3.1 Conceitos Básicos 20
3.2 Modelos 21
3.2.1 Boleano 21
3.3 Indexação 23
3.3.1 Arquivo Invertido 23
3.4 Textos e Formatos 25
3.4.1 Operações em Textos 26
3.4.2 Pré-processando o Documento 26
3.5 Recuperação de Informação em Paralelo e Distribuída 28

4 Algoritmo Seqüencial 31

4.1 Descrição do Algoritmo 31

5 Algoritmo em Paralelo **35**
5.1 Descrição do Algoritmo 36
5.2 Implementação 40
5.3 Consulta em Paralelo 42

6 Conclusão **45**

A Lei de Heaps **51**

B Funções **53**
B.1 Arquivo `fileinverted.c` 53
B.2 Arquivo `stop.c` 55
B.3 Arquivo `strlist.c` 56

Lista de Figuras

2.1 Máquina von Neumann. 6
2.2 Paralelismo lógico (pseudoparalelismo) 6
2.3 Modelo de máquina paralela. Cada nodo compreende uma máquina von Neumann, um processador e memória. 7
2.4 Paralelismo físico 7
2.5 Fração de operações seqüenciais em função do tamanho do problema 11
2.6 Speedup alcançado para diferentes tamanhos de um problema . . . 11
2.7 Modelo PRAM 13

3.1 Os três conectivos para uma consulta $[q = K_a \wedge (K_b \vee \neg K_c)]$. [BYRN99] 22
3.2 Visão lógica do documento [BYRN99]. 27

4.1 Modelo da Lista de Adjacência Encadeada 34

5.1 Entrada do dicionário estendido para um documento particionado. 36
5.2 Construção do vocabulário global para cinco processadores. . . . 38
5.3 Processo de distribuição para sete processadores. 39

A.1 A esquerda, distribuição da freqüência das palavras. A direita, o tamanho do vocabulário. [BYRN99] 51

B.1 Estrutura do Arquivo Invertido 55

Lista de Tabelas

2.1 Implementações do MPI 16

3.1 Texto do arquivo invertido. 24
3.2 Arquivo Invertido. 25

4.1 Números de *stopwords*. * A letra 'c' não foi incluída, devido a linguagem C. 32

Capítulo 1

Introdução

Nos últimos anos, os estudos na área de Recuperação de Informação (RI) ganharam grande importância, devido ao enorme conjunto de informações que se vem produzindo. Um conjunto de informações é comumente chamado de coleção de documentos. Atualmente, a *World Wide Web* (WWW), tem se caracterizado como um dos maiores mecanismos de disseminação de informação, permitindo às pessoas armazenar e pesquisar uma grande quantidade de informações. Isso faz com que seu número cresça assustadoramente e de forma desordenada. Essa explosão de conteúdo, hoje tão visada pela pesquisa, já era apontada por Vannevar Bush na década de 40 [Les] e é em volta deste ponto que gira o próprio núcleo da Recuperação de Informação.

O principal objetivo da área de Recuperação de Informação é estudar algoritmos e estruturas de dados para construir sistemas de Recuperação de Informação. A partir de uma consulta do usuário, o sistema deve retornar de forma rápida os documentos mais relevantes. A ênfase é na recuperação de informação e não na recuperação de dados, mais estudada tradicionalmente na área de banco de dados.

Dois dos grandes problemas enfrentados pelos sistemas de Recuperação de Informação são a consulta e a indexação realizadas sobre grandes coleções de documentos. O custo de indexação e busca cresce juntamente com o tamanho da coleção: grandes coleções invariavelmente resultam em longo tempo de resposta. Para suportar a demanda dos sistemas de Recuperação de Informação neste novo ambiente algoritmos e arquiteturas estão sendo utilizados para explorar o processamento paralelo e distribuído. Os principais estudos dos sistemas paralelos e distribuídos para RI abordam a geração paralela de índices, consulta em base de dados distribuídas, arquiteturas MIMD e particionamento da coleção.

Vários trabalhos têm explorado os métodos de indexação e consulta de forma paralela. Em [JO95], Jeong e Omiecinsky propõe dois diferentes esquemas para particionar um sistema de arquivo invertido numa máquina multi-processador com múltiplos discos, onde tudo é compartilhado, sendo que todo processador pode acessar a memória comum e qualquer disco com o mesmo tempo de acesso.

Em [MZ], Moffat aborda compressão em grandes quantidade de dados e um método de indexação para reduzir o custo de processamento de uma consulta.

Mais recentemente, as pesquisas se voltaram para o paralelismo em redes locais [BCCG95], [TGM93a]. Em [BCCG95], Burkowski examina o problema de desempenho entre o processamento da consulta e a recuperação da informação e estuda a questão da organização física de documentos e índices. Garcia-Molina, em [TGM93a], complementa o trabalho de Burkowski, se concentrando na organização física do índice. Tomasic e Garcia-Molina continuam a pesquisa iniciada em [TGM93a], estudando ainda a relação entre a distribuição dos índices e sua atualização incremental [Bar97].

Este trabalho foi organizado da seguinte forma: o Capítulo 2 apresenta os principais conceitos de computação paralela, necessários para se entender o funcionamento de um algoritmo paralelo. O Capítulo 3 expõe os conceitos básicos de Recuperação de Informação, como modelos de consulta, indexação, pré-processamento em textos, arquivo invertido e RI em paralelo e distribuída. No Capítulo 4 um algoritmo seqüencial é apresentado. No Capítulo 5 é apresentado um algoritmo paralelo, o qual é composto de três fases, sendo que a primeira fase foi implementada neste trabalho. Por fim, as conclusões são apresentadas Capítulo 6.

Capítulo 2

Computação Paralela

Os sistemas de computação paralela e distribuída são compostos por vários processadores que operam concorrentemente, cooperando na execução de uma determinada tarefa. Estes sistemas são necessários quando as tarefas exigem um alto poder computacional e um melhor desempenho na solução dos problemas.

Com o avanço da computação e o uso cada vez maior do computador muitas aplicações têm se tornado complexas, seja no volume de dados manipulados, seja na complexidade de processamento que lhes está associada, sendo que, cada vez mais, os usuários exigem resultados em tempo mais rápido.

O uso de sistemas de computação paralela e distribuída se faz necessário quando os problemas excedem os limites físicos e algorítmicos da computação seqüencial. Com o uso das arquiteturas paralelas objetiva-se aumentar a capacidade de processamento de dados, utilizando o potencial oferecido por um grande número de computadores e processadores.

2.1 Modelos de Arquitetura Paralela e Distribuída

A existência de múltiplas variantes de arquiteturas seqüenciais e paralelas, gerou a necessidade de serem classificadas. Em [Fly72], Flynn baseiou-se em dois conceitos para classificar tais arquiteturas – fluxo de instrução e fluxo de dados. Segundo [Tan97], um fluxo de instruções corresponde a um contador de programa, um sistema com n CPUs possui n contadores de programa, e então n fluxos de instruções. Ainda em [Tan97], um fluxo de dados consiste em um conjunto de operandos, por exemplo, um programa que calcula a média de uma lista de temperaturas possui um fluxo de dados, um programa que calcula a média das tempe-

raturas de cem termômetros espalhados pelo mundo possui cem fluxos de dados.

Segundo a classificação de Flynn, distinguem-se quatro modelos de arquiteturas: SISD, SIMD, MISD, MIMD. Resume-se, a seguir, as características essenciais de cada modelo.

Arquiteturas *Single Intruction Single Data* – SISD

As arquiteturas SISD correspondem, na realidade, às arquiteturas baseadas na máquina de von Neumann, isto é, os computadores seqüenciais convencionais, com o programa armazenado em memória. Nesta classe, estão incluídas todas as arquiteturas que, em cada instante, estão executando apenas um fluxo de instruções e um fluxo de dados, realizando uma operação de cada vez.

Arquiteturas *Single Intruction Multiple Data* – SIMD

Nas arquiteturas SIMD, uma única unidade de controle executa uma instrução de cada vez, mas elas possuem múltiplas unidade aritmética lógica para executar esta instrução em múltiplos conjuntos de dados simultaneamente.

Arquiteturas *Multiple Instruction Single Data* – MISD

As arquiteturas MISD são uma arquitetura estranha, com múltiplos fluxos de instruções operando sobre a mesma porção de dados. Esta arquitetura é bastante discutível, quanto à inclusão de qualquer máquina neste modelo [Tan97].

Arquiteturas *Multiple Intruction Multiple Data* – MIMD

Nesta classe estão incluídas todas as máquinas que, tendo várias unidades processadoras, permitem, em cada instante, a execução de múltiplas instruções diferentes sobre outros múltiplos conjunto de dados. Dos dois tipos de arquiteturas paralelas, SIMD e MIMD, estas são, sem dúvida, mais genéricas e polivalentes, sendo também as de maior divulgação. A maioria dos processadores paralelos estão classificados na categoria MIMD [Tan97].

2.2 Principais Conceitos

Para um melhor entendimento sobre computação paralela, alguns conceitos serão apresentados a seguir. Grande parte destes conceitos podem ser encontrados em [Qui94].

Programas *vs* Processos

Programas e processos são conceitos distintos que ocasionalmente são confundidos. Os programas referem-se ao código-fonte e possuem caráter estático. Já os processos são os programas em execução, com caráter dinâmico, com um fluxo de controle seqüencial e seu espaço de endereçamento.

Um programa pode disparar vários processos durante a sua execução e quando o mesmo programa é executado por dois usuários distintos são gerados dois processos independentes.

Processamento Paralelo

O processamento paralelo é o processamento da informação que enfatiza a manipulação concorrente de dados pertencentes a um ou mais processos de um mesmo problema. O **computador paralelo** é um computador com múltiplos processadores capazes de realizar processamento paralelo.

Modelo de von Neumann

Um computador seqüencial, também conhecido como máquina von Neumann, compreende uma unidade de processamento central (CPU) ligada a uma unidade de armazenamento (memória), como mostra a Figura 2.1. A CPU executa um programa que especifica a seqüência de operações de leitura e escrita na memória. Ao longo dos anos, este modelo, tornou possível os estudos de algoritmos e linguagens de programação, em larga escala, sem a dependência de arquitetura. Com este modelo, os algoritmos podem ser desenvolvidos para uma máquina abstrata de von Neumann, possibilitando aos algoritmos sua execução na maior parte dos computadores com razoável eficiência.

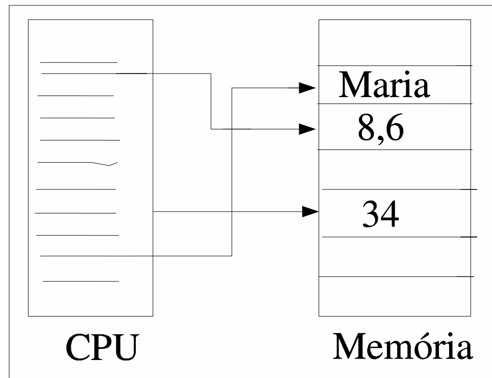

Figura 2.1: Máquina von Neumann.

No modelo seqüencial a concorrência entre os processos é lógica, isto é, o tempo de processamento de cada processo é dividido pelo sistema operacional. Apenas um processo é executado por vez pelo processador, ilustrado na Figura 2.2. Isso faz com que o usuário tenha a impressão de que os processos estão sendo executados em paralelo.

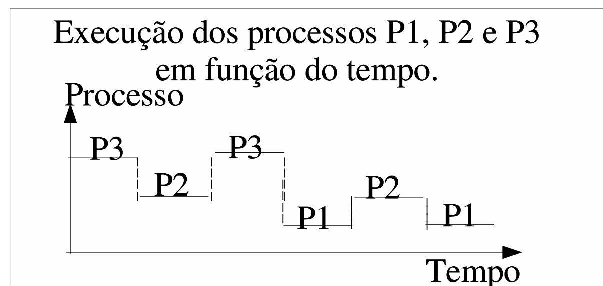

Figura 2.2: Paralelismo lógico (pseudoparalelismo)

Modelo Paralelo

Um modelo de máquina paralela pode ser compreendido como um multi-computador, o qual é formado por um certo número de máquinas von Neumann, ou nodos, interligados por uma rede de interconexão, veja a Figura 2.3. Cada máquina

executa seu próprio programa. Podendo fazer acesso a memória local, enviar e receber mensagens através da rede. As mensagens são usadas para comunicação com as outras máquinas. Numa rede ideal o custo de enviar uma mensagem entre dois nodos é independente da localização dos nodos e do tráfego na rede, mas depende do tamanho da mensagem [Fos95].

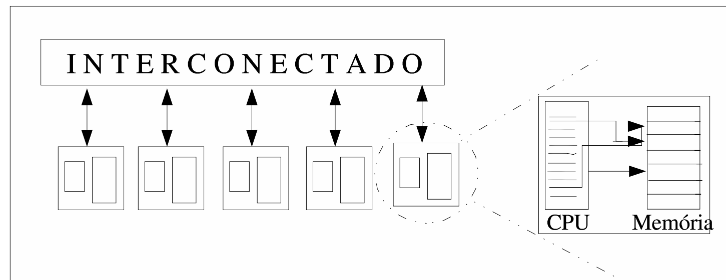

Figura 2.3: Modelo de máquina paralela. Cada nodo compreende uma máquina von Neumann, um processador e memória.

Desta forma, a concorrência entre os processos num computador paralelo é real, visto que cada computador está executando um processo diferente simultaneamente, com pode-se obeservar na Figura 2.4

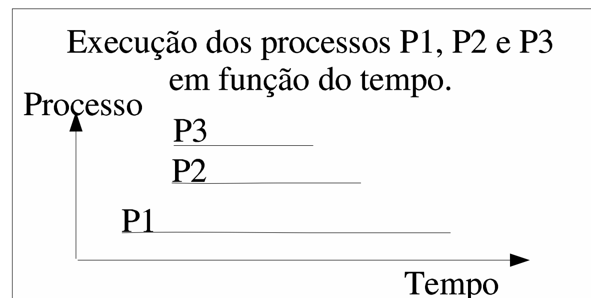

Figura 2.4: Paralelismo físico

2.3 Tipos de Paralelismo

Existem dois modelos de paralelismo na computação paralela: o paralelismo de dados e o paralelismo de controle.

O **paralelismo de dados** faz o aproveitamento da concorrência que deriva da aplicação da mesma operação a múltiplos elementos de uma estrutura de dados. Teoricamente $k - unidades$ produzem um aumento de vazão (número de resultados produzidos por unidade de tempo) de $k - vezes$ no sistema.

Neste tipo de paralelismo os processadores executam as mesmas instruções sobre dados diferentes. Sendo este estilo de paralelismo aplicado, por exemplo, na resolução de sistemas lineares e multiplicação de matrizes, ou somar o valor 20 a todos os elementos de uma lista.

Em contraste ao paralelismo de dados, há o **paralelismo de controle**, onde diferentes operações são aplicadas sobre diferentes elementos de dados simultaneamente. O fluxo de dados sobre estes processos pode ser arbitrariamente complexo.

No desenvolvimento de algoritmos paralelos, deve-se procurar escolher o estilo de paralelismo que mais se adapte à natureza do problema. Muitos problemas reais também podem explorar ambos os tipos de paralelismo.

2.4 Escalabilidade e Concorrência

Grande parte dos algoritmos existentes foram projetados para computadores com um único processador, esta situação faz com que novos algoritmos sejam estudados e desenvolvidos para que sejam capazes de realizar múltiplas operações simultaneamente. No futuro, talvez, os programas deverão ser capazes de explorar os múltiplos processadores localizados em cada computador e os processadores disponíveis através de uma rede. A **concorrência** torna-se um requisito fundamental dos algoritmos e programas.

Além disso, o resultado produzido por um programa não pode depender do local onde as tarefas estão sendo executadas. Os algoritmos devem ser projetados e implementados sem levar em consideração o número de processadores com o qual serão executados. Esta é uma forma direta de atingir a noção de **escalabilidade**, na medida em que o número de processadores aumenta, o número de tarefas por processador é reduzido, mas o algoritmo, ele próprio, não necessita ser modificado.

A escabilidade é tão importante para a programação paralela quanto a porta-

bilidade é para a programação seqüencial, como forma de proteger o investimento em *software*. Um programa que é capaz de ser executado apenas em um número fixo de processadores é um “mau programa”, assim como, o programa que só pode ser executado em uma única plataforma.

Sendo assim, a concorrência refere-se a habilidade de efetuar várias ações simultaneamente; isto é essencial se um programa for executado em mais que um processador. E a escalabilidade algoritmica é quando o nível de paralelismo cresce pelo menos linearmente com o tamanho do problema. E uma arquitetura é escalável se ela continua rendendo o mesmo desempenho por processador quando o número de processadores aumenta.

As escalabilidades algorítmica e arquitetural são importantes, porque elas permitem a resolução de grandes problemas na mesma quantidade de tempo ao se usar um computador paralelo com mais processadores.

Algoritmos paralelos em dados são mais escaláveis que algoritmos paralelos em controle, pois o nível de paralelismo de controle é geralmente uma constante, independente do tamanho do problema, enquanto que o nível de paralelismo de dados é uma função crescente do tamanho do problema.

2.5 Speedup

Existe duas importantes medidas de qualidade de algoritmos paralelos, são *speedup* e eficiência.

O ***speedup*** é definido como: a razão entre o tempo em que um computador paralelo executa o algoritmo seqüencial mais eficiente (com apenas um processador) e o tempo em que o mesmo computador paralelo executa o correspondente algoritmo paralelo usando p processadores, ambos sobre a mesma computação. E a **eficiência** de um algoritmo executado em p processadores é o *speedup* dividido por p.

Ou seja,

$$Speedup = \frac{tempo\,seq\ddot{u}encial(1\,processador)}{tempo\,paralelo(p\,processadores)}$$

e,

$$Efici\hat{e}ncia = \frac{Speedup}{p}$$

O *Speedup* ideal a ser alcançado, isto é, o máximo ganho obtido com a paralelização, deveria assintotar ao número de processadores utilizados, entretanto na prática isto não acontece.

O *Speedup* não é diretamente proporcional ao número de processadores utilizados. Contudo, afirma-se que um *Speedup Superlinear* (maior que linear) é possível desde que a escolha do algoritmo seja feita antes da instância do problema e que o algoritmo paralelo trate alguns casos específicos que o seqüencial trata de modo genérico [Qui94].

2.6 Lei de Amdahl

A Lei de Amdahl é uma maneira de expressar o *Speedup* máximo que pode ser alcançado, utilizando p processadores em um algoritmo paralelo que possui uma fração $0 \leq f \leq 1$ de código que deve ser, obrigatoriamente, executado seqüencialmente.

Lei de Amdahl Seja f a fração de operações numa computação que deve ser executada seqüencialmente, onde $0 \leq f \leq 1$. O *Speedup* máximo (S) alcançado por um computador paralelo com p processadores executando a computação é

$$S \leq \frac{1}{f + \frac{(1-f)}{p}}$$

De acordo com a lei de Amdahl uma fração de operações seqüenciais, mesmo que em números pequenos, pode significativamente limitar o *Speedup* alcançado por um computador paralelo. Por exemplo, se um algoritmo paralelo possui uma parcela de 10% de operações que necessitam ser executadas seqüencialmente, o *speedup* máximo que pode ser alcançado é igual a 10, não importando a quantidade de processadores usados. Entretanto, essa lei não contempla certas aplicações paralelas, onde a fração seqüencial do algoritmo é reduzida à medida que o tamanho do problema aumenta.

Sob essas circunstâncias a Lei de Amdahl não é um bom indicador de *Speedup*, pois ela assume que a fração seqüencial é fixa, o que não é verdade. E como pode ser visto na Figura 2.5, em geral, quando o tamanho do problema cresce a fração f de operações seqüenciais decresce, tornando o problema mais viável à paralelização. Este fenômeno é chamado de Efeito Amdahl [Goo77]. Isto pode ser confirmado na Figura 2.6, que exibe o *Speedup* obtido com o aumento do tamanho de um problema. As Figuras 2.5 e 2.6 são de um exemplo extraído de [Qui94].

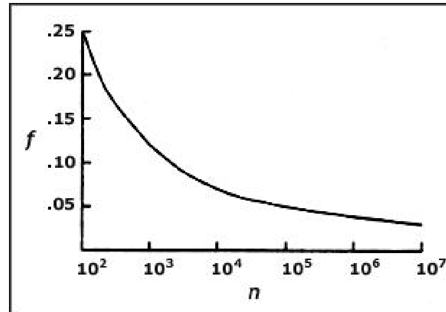

Figura 2.5: Fração de operações seqüenciais em função do tamanho do problema

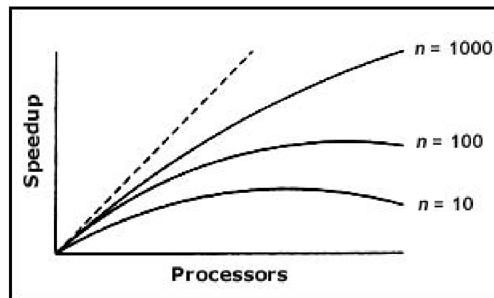

Figura 2.6: Speedup alcançado para diferentes tamanhos de um problema

Apenas em 1988, é que John Gustafson e Ed Barsis apresentaram uma outra lei – *Revaluating Amdahl's Law. John L. Gustafson. Em Communications of the ACM 31, n.5, pp. 532-533, 1988* – provando que a lei de Amdahl não poderia ser sempre aplicada, pois considerava que as partes seqüenciais e paralelas eram independentes entre si, o que não é verdade. Eles definiram:

$$T(p) = f + (1 - f) = 1, \text{ e}$$
$$T(1) = f \times (1 - f) \times p,$$

onde f é a parte do programa que será executada seqüencialmente; e $T(p) = 1$, significa que o problema foi projetado para computação paralela. Substituindo na fórmula para o cálculo do *speedup*, temos a Lei de Gustafson-Barsis com a seguinte definição:

$$S = p - (p - 1) \times f$$

Essa lei alterou completamente o panorama das pesquisas com sistemas paralelos e distribuídos, impulsionando o seu estudo, pois provou que estes sistemas poderiam apresentar ganhos maiores do que o imaginado, veja um exemplo abaixo: Tendo $f = 0,05$ e $p = 10$, tem-se que, a Equação (2.1) é usada a fórmula de Amdahl e na Equação (2.2) é usada a fórmula de Gustafson e Barsis.

$$S \leq \frac{1}{0,05 + \frac{(1-0,05)}{10}} \quad \leq \frac{1}{0,05 + 0,095} \quad \leq \frac{1}{0,145} \quad \leq 6,896 \tag{2.1}$$

$$S = 10 - (10 - 1) \times 0,05 \quad S = 10 - 0,45 \quad S = 9,55 \tag{2.2}$$

o *speedup* ideal seria 10, notadamente, pela fórmula de Gustafson e Barsis obtêm-se um resultado bem melhor.

No entanto, a lei de Amdahl pode ser relevante quando os programas seqüenciais são paralelizados incrementalmente. Nesta abordagem no desenvolvimento de programas paralelos, um programa seqüencial é inicialmente analisado para identificar os componentes com requisitos computacionais. Esses componentes são depois adaptados para execução paralela, um após o outro, até atingir um rendimento aceitável. Conforme [Fos95], a lei de Amdahl aplica-se claramente nesta situação, porque os custos da computação dos componentes que não são paralelizáveis determinam o limite inferior do tempo de execução do programa paralelo. Por isso, esta estratégia de paralelização, "parcial" ou "incremental", é geralmente efetiva apenas em pequenos computadores paralelos. A lei também pode ser útil quando se analisa o rendimento de programas com paralelismos nos dados, em que alguns componentes podem não ser adaptáveis a uma formulação de paralelismo nos dados.

2.7 Modelo PRAM

O modelo de computação paralela PRAM (*Parallel Random Acsess Machine*) é uma extensão do modelo seqüencial RAM e o mais conhecido modelo de computação paralela. Este modelo permite aos desenvolvedores tratarem o poder de processamento como um recurso ilimitado.

O modelo PRAM não é um modelo real, pois ele ignora a complexidade de comunicação entre os processadores, não havendo nenhum custo na comunicação.

Ao desenvolver um algoritmo PRAM a preocupação é focada apenas no paralelismo inerente à computação, pois a complexidade de comunicação não é considerada. Para alguns algoritmos, soluções PRAM de custo ótimo significam que o número total de operações efetuadas pelo algoritmo PRAM é da mesma complexidade que um algoritmo seqüencial equivalente. Um algoritmo PRAM de custo ótimo é um dos principais candidatos para servir como base para eficientes algoritmos em computadores paralelos reais.

O modelo PRAM pode ser descrito como sendo um conjunto ilimitado de processadores, no qual cada um possui um índice único e uma memória local própria, podendo comunicar-se com os demais processadores por meio de uma memória global compartilhada, com uma unidade de controle. Esta forma de comunicação possibilita o acesso (leitura ou escrita) simultâneo de vários processadores a uma mesma posição da memória global, o modelo está ilustrado na Figura 2.7.

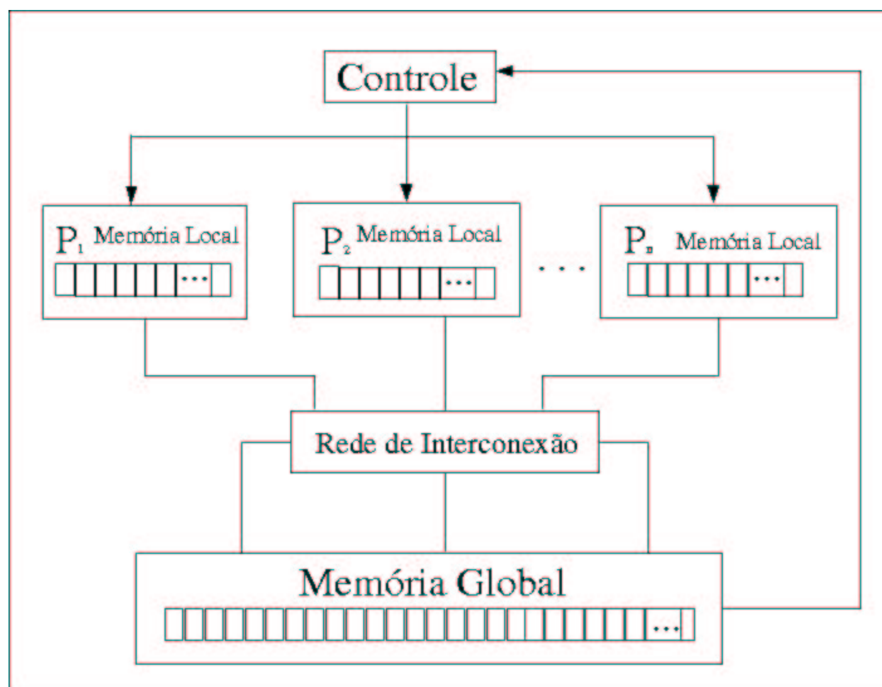

Figura 2.7: Modelo PRAM

Uma computação PRAM inicia com a entrada armazenada em memória global e um único processador ativo. Durante cada passo da computação um processador

ativo pode ler um valor de sua memória local ou global, executar uma operação e escrever o resultado na memória local ou global. Durante a computação, um processador pode ativar um outro processador. Todos os processadores ativos devem executar a mesma instrução, embora em posições de memória diferentes. A computação termina com o fim do processamento do último processador.

Definição O custo de uma computação PRAM é o produto da complexidade de tempo paralela e o número de processadores usados. Por exemplo, um algoritmo PRAM que tenha complexidade de tempo $\Theta(logp)$ usando p processadores tem custo $\Theta(plogp)$.

2.8 Ambiente Paralelo em Sistemas Distribuídos

Os sistemas computacionais distribuídos aplicados à computação paralela permitem uma melhor relação custo/benefício para a computação paralela. Esses sistemas oferecem a potência computacional adequada às aplicações que não necessitam de uma máquina maciçamente paralela, porém necessitam de uma potência computacional maior que uma máquina seqüencial pode oferecer [Cas99].

Para a realização da computação paralela sobre sistemas computacionais distribuídos são utilizados ambientes de troca de menssagens, que têm sido constantemente aperfeiçoados, permitindo assim a união de uma quantidade cada vez mais significativa de computadores. As vantagens dos sistemas distribuídos são basicamente:

- redução de custo com a utilização de *hardware* existente;
- aumento de desempenho pela atribuição de tarefas de acordo com a arquitetura apropriada;
- exploração da heterogeneidade natural das aplicações;
- e a utilização de recursos conhecidos, fornecidos pelas máquinas virtuais.

Porém vários tipos de incompatibilidades podem ocorrer nos sistemas distribuídos heterogêneos [Beg94]: arquiteturas, formato de dados, potência computacional, carga de trabalho em cada máquina e carga de trabalho na rede.

PVM (*Parallel Virtual Machine*) e MPI (*Message Passage Interface*) são exemplos de ambientes paralelos virtuais, ambos são por troca de mensagem.

2.9 MPI

No modelo de programação MPI, a computação consiste de um ou mais processos que se comunicam por meio de funções que enviam e recebem mensagens de outros processos.

Nas implementações MPI, um conjunto fixo de processos é criado na inicialização do programa, e um processo é criado por processador. Contudo, estes processos podem executar diferentes programas. Conseqüentemente, o modelo de programação MPI é algumas vezes referido como MPMP (*Multiple Program Multiple Data*) para distinguir do modelo SPMP (*Simple Program Multiple Data*) em que cada processador executa o mesmo programa. Sua biblioteca pode ser usada nas linguagens C, C++ e Fortran. MPI é composto apenas de 129 funções, muitas das quais são variantes das outras ou simétricas, como descrito em [MS96] que oferecem os seguintes serviços:

- Comunicação ponto-a-ponto — implementado diversos tipos de serviços;
- Comunicação coletiva — implementado diversos tipos de serviços;
- Suporte para grupos de processos — os processos são relacionados em grupos, os quais, são identificados pela sua classificação dentro do seu grupo, a partir do número zero;
- Suporte para grupos de comunicação — podem ser definidos como escopos que relacionam um determinado grupo de processos. Estes tipos de instâncias são implementadas com o intuito de garantir que não existam mensagens que sejam recebidas ambiguamente por grupos de processos não relacionados;
- Suporte para topologia de processos — onde são fornecidas primitivas que permitem ao programador definir a estrutura topológica que descreve o relacionamento entre processos.

De acordo com [Cas99], o padrão MPI apresenta as seguintes características:

Eficiência: foi cuidadosamente projetado para executar eficientemente em máquinas diferentes. Especifica somente o funcionamento lógico das operações. Deixa em aberto a implementação. Os desenvolvedores otimizam o código usando características específicas de cada máquina.

Facilidade: define uma interface não muito diferente dos padrões PVM, *Express*, P4, etc, e acrescenta algumas extensões que permitem maior flexibilidade.

Portabilidade: é compatível com sistemas de memória distribuída, *shared-memory* e NOWS (*Network of Workstations*).

Transparência: permite que um programa seja executado em sistemas heterogêneos sem mudanças significativas.

Segurança: provê uma interface de comunicação confiável. O usuário não precisa se preocupar com falhas na comunicação.

Escalabilidade: o MPI permite crescimento em escala sob diversas formas, por exemplo, uma aplicação pode criar subgrupos de processos que permitem operações de comunicação coletiva para melhorar o alcance dos processos.

As principais implementações MPI são apresentadas na Tabela 2.1.

Tabela 2.1: Implementações do MPI

IBM MPI	Implementação IBM para Clusters.
MPICH	*Argone National Laboratory; Mississipi State University*
UNIFY	*Mississipi State University*
CHIMP	*Edinburgh Parallel computing University*
LAM	*Ohio Supercomputer Center*
PM PIO	NASA
MPIX	*Mississipi State University NSF Enginnerring Research Center.*

Para este trabalho foi utilizada a implementação LAM (*Local Area Multicomputer*). Esta implementação é portável à maioria das máquinas UNIX.

LAM implementa todo o padrão MPI-1, e muitas características do padrão MPI-2. Possui tolerância a falha e rápida comunicação. Além disso, esta implementação também facilita o trabalho dos desenvolvedores de algoritmos paralelos, pois pode ser instalada em uma máquina seqüencial com o sistema operacional Linux, onde pode ser simulado o funcionamento de algoritmos paralelos. A implementação LAM/MPI é gratuita, sob licença BSD, e está disponível em [ALB96].

Capítulo 3

Recuperação de Informação

Ao longo da história da humanidade, sem exceção, a pessoa mais rica do mundo sempre comandou os recursos minerais: a terra, o ouro, o petróleo. Hoje, um dos homens mais rico do mundo não tem terra, não tem ouro, não tem petróleo, não tem prédios, não tem máquinas. Pelo fato de controlar o processo de conhecimento faz dele a pessoa mais rica do mundo e de sua empresa a mais valiosa. Pela primeira vez, em toda a história da humanidade, é possível ser incrivelmente rico graças ao controle do conhecimento.

Nos dias de hoje, a informação que gera conhecimento é tida como um dos mais importantes meios de ter sucesso profissionalmente. A revolução da Tecnologia da Informação junto com a Internet contribui de forma contundente para que o volume de informações cresça assustadoramente. Informações estas, que encontram-se acessíveis através da *World Wide Web* (WWW), mas sem nenhuma, ou pouca, estrutura e mal organizada, formando uma verdadeira "Torre de Babel". A WWW, em 1999, possuía por volta de 200 milhões de páginas, quase 50 gigabytes de dados, além disso, cresce a uma taxa exponencial [BYRN99]. A maior parte destas informações é constituída na forma de textos, tais como, livros, periódicos, trabalhos científicos, etc. Para recuperar uma informação que se deseja nesta base de dados é importante pré-processar o texto e construir um índice que permita efetuar pesquisas de forma rápida e eficiente.

A conceituação de Recuperação de Informação, definida inicialmente por Calvin Mooers em [Moo51] e [Fra92], vem dada de uma forma eminentemente funcional e não descritiva:

> "Recuperação de informação é o nome do processo ou método onde um posssível usuário de informação pode converter a sua necessidade

> de informação numa lista real de citações de documentos armazenados que contenham informações úteis a ele ... recuperação de informação abrange os aspectos intelectuais da descrição da informação e a sua especificação para busca, assim como também quaisquer sistemas, técnicas ou máquinas que sejam empregadas para efetuar a operação" (Mooers, 1951 apud Saracevic, 1995).

3.1 Conceitos Básicos

Os conceitos apresentados neste capítulo e a definição dos vários termos usados podem ser encontrados em [BYRN99].

A recuperação de uma informação relevante é diretamente influenciada pela consulta do usuário, bem como, pela visão lógica dos documentos adotados pelo sistema de recuperação.

O usuário de um sistema de recuperação de informação tem que traduzir o assunto em que está interessado a pesquisar numa linguagem fornecida pelo sistema. Na maioria das vezes isto implica em especificar um conjunto de palavras que resume um assunto que transmita a semântica da informação desejada.

Os modelos clássicos de Recuperação de Informação consideram que cada documento seja descrito por um conjunto de palavras-chave, comumente chamados de termos indexados. Um *termo indexado* é simplesmente uma palavra cuja semântica ajuda a lembrar do tema principal do documento. Desta forma, termos são usados para indexar e sumarizar o conteúdo do documento. Geralmente, os termos indexados são substantivos devido a estes possuírem significado próprio e uma semântica fácil de ser identificada e compreendida. Os termos são diretamente extraídos de uma coleção de documentos. Esse modelo de representação das palavras-chave fornece a visão lógica dos documentos.

Nem todos os termos do conjunto de termos indexados de um documento possuem o mesmo peso para descrever o conteúdo do documento. Decidir a importância de um termo indexado para resumir o assunto de um documento é uma tarefa não trivial. Mesmo assim, existem propriedades de um termo que são fáceis de medir e úteis para avaliar o potencial de um termo como tal. Por exemplo, considere uma palavra que aparece em todos os documentos de uma coleção com milhões de documentos, essa palavra é completamente inútil para ser um termo indexado, pois não acrescenta nada a uma consulta sobre um documento em que o usuário possa estar interessado. Por outro lado, imagine uma palavra que aparece em apenas dez documentos, essa pode ser muito útil porque limita consideravel-

mente o espaço do documento que deve ser de interesse do usuário.

3.2 Modelos

Ribeiro-Neto e Baeza-Yates em [BYRN99] abordam três modelos clássicos de recuperação de informação: Modelo Booleano, Modelo Vetorial e o Modelo Probabilístico. No modelo Booleano, os documentos e consultas são representados como um conjunto de termos indexados. No modelo vetorial, os documentos e consultas são representados como vetores em um espaço t-dimensional. No modelo probabilístico, a estrutura para modelar a representação dos documentos e consultas é baseado na teoria da probabilidade. Os modelos também são referenciados como conjunto teórico, algébrico e probabilístico, respectivamente.

3.2.1 Boleano

O modelo Booleano é baseado na teoria dos conjuntos e na álgebra de Boole. O conceito de conjunto é totalmente intuitivo, o modelo Booleano fornece uma estrutura de fácil compreensão para usuários comuns de sistemas de RI. As consultas são especificadas usando expressões booleanas. Dada a simplicidade natural e um formalismo claro, o modelo, recebeu grande atenção e foi adotado por muitos sistemas comerciais bibliográficos.

O critério do modelo Booleano é baseado no sistema binário, ou o termo da consulta está presente no conjunto de termos indexados ou o termo não está presente. Uma consulta q é composta dos termos ligados a três conectivos: `AND`, `NOT`, `OR`. Uma consulta é uma expressão booleana que pode ser representada como uma disjunção de vetores conectivos (isto é, na forma normal disjuntiva – DNF). Por exemplo, a consulta $[q = K_a \wedge (K_b \vee \neg K_c)]$ pode ser escrita na forma normal disjuntiva como $[\vec{q}_{dnf} = (1,1,1) \wedge (1,1,0) \vee (1,0,0)]$, onde cada um dos elementos é um vetor binário com um determinado peso associado a tupla (K_a, K_b, K_c). Estes vetores são chamados de elementos conectivos de $\vec{q}_{dnf}$. A Figura 3.1 ilustra os três componentes conectivos para a consulta q.

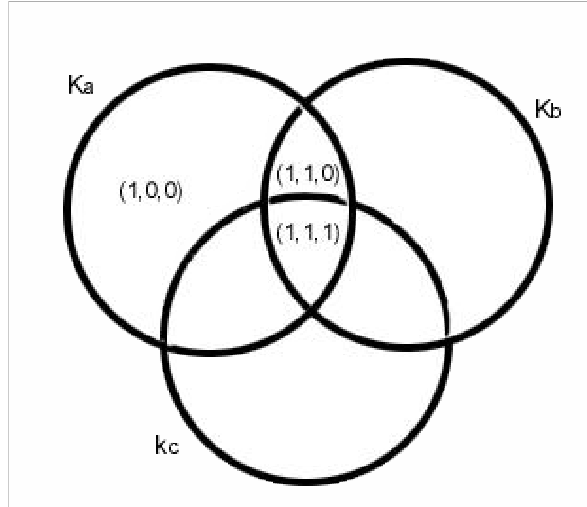

Figura 3.1: Os três conectivos para uma consulta $[q = K_a \wedge (K_b \vee \neg K_c)]$. [BYRN99]

Definição Para o modelo Booleano, as variáveis dos termos indexados são todas binárias isto é, $w_{i,j} \in \{0, 1\}$. Uma consulta d é uma expressão boolena convencional. Sendo $\vec{q}_{dnf}$ a forma normal disjuntiva para uma consulta q. Além disso, sendo $\vec{q}_{cc}$ ser qualquer elemento conjuntivo de $\vec{q}_{dnf}$. A similaridade de um documento d_j para a consulta q está definida como:

$$\mathbf{sim(d_j, q)} = \begin{cases} 1 \text{ se } \exists \vec{q}_{cc} | (\vec{q}_{cc} \in \vec{q}_{dnf}) \wedge (\forall_{ki}, g_i(\vec{d_j}) = g_i(\vec{q}_{cc})) \\ 0 \text{ caso contrário} \end{cases}$$

Se $sim(d_j, q) = 1$ então o modelo Booleano prevê que o documento d_j é relevante para a consulta q (ele pode não ser). Caso contrário, a previsão é que o documento não é relevante.

No modelo Booleano, não existe um grau intermediário de relevância para o documento em relação à consulta, ele é relevante ou não.

Michard em [Mic82] e Young em [YS93], apontam mais desvantagens que vantagens ao modelo Booleano. Os estudos têm mostrado que os usuários têm dificuldade em expressar suas consultas no formato Booleano. Estudos em [Mic82], mostram que para usuários inexperientes, usar o conectivo AND implica em aumentar o escopo da consulta, quando é justamente ao contrário. A principal desvan-

tagem apontada por [BYRN99] é que muitos ou poucos documentos são recuperados numa consulta.

Apesar disso, o modelo Booleano é mais simples de ser implementado e apresenta um melhor tempo de busca, sendo utilizado pela maioria das máquinas de busca da WWW.

3.3 Indexação

Comprimir as informações de uma base de dados é apenas parte da solução para lidar com o grande volume de informações. A compressão não soluciona o problema de como as informações devem ser organizadas nem como elas podem ser consultadas de forma eficiente. Por isso, é necessário indexá-las.

A indexação, aqui, é muito similar a indexação que a maioria das pessoas estão acostumadas, o uso de um índice num livro. Por exemplo, com um livro em mãos e a necessidade de encontrar o tópico desejado, a primeira coisa a fazer é olhar no sumário ou no índice alfabético, para ver onde se encontra a página referente ao assunto desejado. Através do índice é possível localizar a informação sem a necessidade de olhar "página por página". Um livro sem sumário e índice pode causar uma grande frustação. O mesmo pode ocorrer com um sistema de reuperação de informação. A maioria dos sistemas usam índices para encontrar palavras de uma consulta.

A técnica mais usada para indexação pelos sistemas de busca é o arquivo invertido, devido ao fato dele possuir baixo custo de construção e manutenção. Em princípio um arquivo invertido de um texto de n caracteres pode ser feito em tempo $O(n)$ [Fra92]. Na próxima seção discutimos sobre o arquivo invertido.

3.3.1 Arquivo Invertido

Um arquivo invertido (ou índice invertido) é um mecanismo para indexação de coleções de textos orientado à palavra. O arquivo invertido é composto por um vocabulário – lista de todos os termos que aparecem em uma coleção – onde cada termo do vocabulário possui um ponteiro para todas as ocorrências do termo no texto principal, sendo que cada ponteiro, é, de fato, o número do documento em que o termo aparece, conforme descrito por [IHWB99], dependendo da granularidade[1], pode ser definido como a posição do termo em um documento. A entrada

[1]Definido como o tamanho do bloco a ser indexado, pode-se guardar em que parte do documento o termo aparece, ou simplesmente referir-se que o termo aparece no documento

do arquivo invertido também é conhecida como *Posting*. Um *posting* pode ser definido como uma tupla (k_i, d_j), onde k_i é o identificador do termo e d_j é o identificador do documento. A seguir é exemplificada a construção de um arquivo invertido, baseada numa coleção de documentos, ilustrada na Tabela 3.1.

Tabela 3.1: Texto do arquivo invertido.

Documento	**Texto**
1	`o homem seria explicado` pelo mesmo `substrato` ou pela mesma `natureza`...
2	o `morto` é `igual` a `nada`, e não `sente nenhuma sensação`, `trata`-se de uma `mudança`...
3	`Homem morto` pela `natureza`.

A Tabela 3.2 mostra o resultado do arquivo invertido após um pré-processamento do texto, os termos do vocabulário estão ordenado lexicograficamente. É conveniente guardar o número de ocorrências do termo num determinado documento. Uma consulta pode ser realizada usando-se qualquer estrutura de dados apropriada, tais como, árvore-B, *hashing*, *tries*[2]. O custo do tempo de busca das duas últimas estruturas é $\mathrm{O}(m)$, onde m é o tamanho do termo a ser pesquisado, independente do tamanho do texto. Contudo, com as palavras indexadas na ordem lexicográfica pode-se realizar uma busca binária em $\mathrm{O}(logn)$, onde n é o tamanho do banco de dados dos termos. A busca binária pode ser realizada em árvores B e *tries*, mas não em *hashing*.

[2] *Tries* é uma árvore de multi-caminhos que armazena um conjunto de *strings*, capaz de recuperar qualquer *string* num tempo proporcional ao seu tamanho (indenpendente do número de *strings* armazenadas)

Tabela 3.2: Arquivo Invertido.

Vocabulário	**Ocorrências**	**Vocabulário**	**Ocorrências**
explicado	Doc 1	nenhuma	Doc 2
homem	Doc 1, Doc 3	sensacao	Doc 2
igual	Doc 2	sente	Doc 2
morto	Doc 2, Doc 3	seria	Doc 1
mudanca	Doc 2	substrato	Doc 1
nada	Doc 2	trata	Doc 2
natureza	Doc 1, Doc 3		

Uma grande vantagem do arquivo invertido é o espaço usado para o armazenamento do vocabulário. Conforme explicado no Apêndice A.1, página 51, a lei de *Heap's* demonstra que o vocabulário cresce em $O(n^{\beta})$. Onde β é uma constante entre 0 e 1, dependendo do texto. Como exemplo, citado, para 1Gb da coleção TREC-2 o vocabulário tem um tamanho de apenas 5Mb.

3.4 Textos e Formatos

O meio de comunicação mais importante ao longo da história foi a escrita, textos são escritos nas mais diversas linguagens e nos mais diversos meios, escritos em pedra, madeira, papel e, atualmente, na forma digital.

Existem os mais diferentes formatos para representar um documento digital. Entre os mais conhecidos estão: `.html`, `.ps`, `.pdf`, `.rtf`, `.txt`, `.doc`, entre outros.

No entanto, alguns formatos dificultam a possibilidade de filtragem de informação, pois, nem tudo que está presente no documento é conteúdo útil, existem muitas *tags* de controle, que são usadas para formar a apresentação do texto na tela do computador. Além disso, existem os formatos que não são de domínio público, ou seja, não há informação disponível para o conhecimento dessas *tags*.

Como o objetivo do trabalho não é fazer filtragem em documentos para obter os termos a serem indexados, o sistema desenvolvido trabalha apenas em textos no formato ASCII, especificamente textos em `.txt`.

3.4.1 Operações em Textos

Nem todas as palavras presentes no texto são significativas para representar sua semântica. Algumas palavras possuem mais significados que outras. Fazendo-se, assim, necessário realizar um pré-processamento sobre o texto, afim de minimizar os termos mais irrelevantes presentes no conjunto de termos indexados. Durante o pré-processamento outras operações podem ser feitas, tal como, elimintação de *stopwords*[3], *stemming*[4], construção de um dicionário de sinônimos, e compressão.

Como o conjunto de termos indexados representará o documento, deve-se ter o cuidado de não indexar palavras que não trazem nenhuma representação semântica ao documento. Por exemplo, o termo "em" não possui nenhum significado e se indexada poderá recuperar vários documentos que não possuem nenhuma relação com o assunto que o usuário deseja. O uso de todas as palavras de um documento para formar o vocabulário trará mais prejuízos que benefícios. Dessa forma, o pré-processamento do texto pode ser visto como um processo de simplificação e controle do tamanho do vocabulário, além de contribuir para um melhor desempenho às consultas.

3.4.2 Pré-processando o Documento

As principais operações para o pré-processamento do texto são:

Análise Léxica – tem o objetivo de eliminar os dígitos, hífens, sinais de pontuação, acentos e conversão de todas as letras para maiúsculas ou minúsculas.

Eliminação de *Stopwords* – objetiva filtrar as palavras que aparecem com muita freqüência e sem significado, a fim de reduzir o tamanho do vocabulário.

Stemming – tem o objetivo de retirar os sufixos e prefixos das palavras, encontrando a forma primitiva da palavra, por exemplo: atualmente, atualizado, atual.

Seleção dos índices – determinar quais palavras, ou grupos de palavras, serão usadas como elementos de indexação. De fato, os substantivos possuem mais semântica que adjetivos, advérbios e verbos.

Dicionário – construir um dicionário de sinônimos para os termos, com o objetivo de expandir uma consulta.

[3]Palavras com pouco significado, tais como, artigos e preposições

[4]Redução de uma palavra a sua raíz gramatical

Uma visão lógica das fases que podem ser aplicadas a um documento pode ser vista na Figura 3.2.

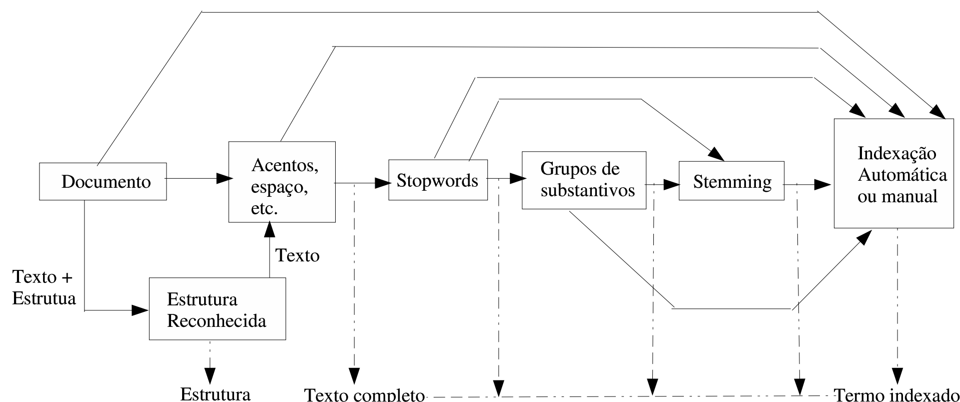

Figura 3.2: Visão lógica do documento [BYRN99].

Análise Léxica

A análise léxica é o processo de conversão de um canal de caracteres (texto de documentos) em um canal de palavras que são candidatas para serem adotadas como termos indexados. Sendo assim, o maior objetivo é a identificação das palavras no texto. Por exemplo, a palavra "**Mão-de-Obra**" seria transformada em: "**mao de obra**", na análise léxica também são descartados números e sinais de pontuação. Alguns sistemas consideram números de dezesseis dígitos, pois podem ser números de cartão de créditos, tornando-se útil a alguns sistemas.

A análise também faz a conversão dos caracteres para maiúsculas ou minúsculas. A aplicação de algumas destas técnicas pode trazer perda de informação, um exemplo é a conversão da palavra **"Estado"** para **"estado"**, ambas possuem significados diferentes, ou a retirada de hífens das palavras, a palavra **"mão-de-obra"** é diferente de **"mao"** e **"obra"**, pois com a eliminação das *stopwords* a preposição **"de"** não é indexada transformando **"mao"** e **"obra"** dois termos distintos.

Eliminação de *stopwords*

Palavras que aparecem freqüentemente na coleção de documentos não têm a necessidade de serem indexadas. Algumas palavras chegam a ocorrer em 80% da

coleção [IHWB99]. Essas palavras são chamadas de *stopwords*, os artigos, preposições e conjunções são sérias candidatas a lista de *stopwords*.

A eliminação das *stopwords* reduz de forma considerável a quantidade de termos indexados. Apenas com seu uso é possível reduzir 40% ou mais o número de termos. A lista de *stopwords* pode ser estendida, incluindo-se outras palavras como verbos, advérbios e adjetivos.

3.5 Recuperação de Informação em Paralelo e Distribuída

Para suportar a complexidade das máquinas de busca e a grande quantidade de variáveis que fazem parte dos modernos ambientes de sistemas de RI, a programação paralela e os sistemas distribuídos surgem como uma nova alternativa para lidar com esses problemas.

Muitos estudos de como indexar, particionar e consultar coleções de documentos foram feitos nos últimos anos. Clarke e Gormack em [CC95] estudaram a indexação dinâmica de arquivos invertidos num sistema de RI distribuído. A estrutura de dados apresentada por eles demonstrou bom desempenho nas operações de consulta, em alguns casos usando apenas um acesso a disco, a atualização do índice não atrapalhou significativamente o desempenho da consulta.

Burkowski [Bur90] estudou a divisão de servidores de texto e máquinas de indexação em vários processadores, concluindo que sistemas que fazem a divisão fornecem um melhor tempo de resposta em relação aos sistemas que trabalham com ambos, servidores de texto e máquinas de indexação.

Tomasic e Garcia-Molina também realizaram uma série de pesquisas na área de RI e sistemas distribuídos. Em [TGM93b] examinaram a alocação de termos indexados para uma coleção de documentos. A alocação de todos os termos, para um documento individual, é melhor em um único processador, ao invés de vários.

Em [TGM93a], Tomasic e Garcia-Molina sugerem que uma organização onde cada máquina contenha a lista invertida dos documentos que residem na máquina local, é melhor, em termos da eficiência da consulta. Para gerar esta última estrutura, bastaria que cada máquina executasse, separadamente, um algoritmo sequencial de geração de lista invertida. Garcia-Molina, em [TGM93a], chama esta organização mais simples de *disk index*, e uma outra, gerada por [JK97], de *system index*.

Em [Bar97] foi realizado o estudo do processamento paralelo de consultas, em um ambiente com memória distribuída, utilizando o modelo vetorial de recuperação de informação. Também foram propostos algoritmos paralelos para a

execução de consultas sobre o modelo vetorial em ambas as organizações (*system index* e *disk index*) e uma avaliação de desempenho destes algoritmos. Ainda são estudadas outras formas de organização das listas invertidas não abordadas na literatura, como por exemplo, o agrupamento dos termos em face de suas co-ocorrências nos documentos (*clustering*). Os experimentos e análises demonstraram que a organização *system index* provê melhor desempenho que o *disk index* para a coleção TREC3, que é uma coleção de 2 GigaBytes.

Capítulo 4

Algoritmo Seqüencial

A linguagem utilizada para o desenvolvimento do projeto foi a linguagem C, utilizando o compilador `gcc` – GNU project C and C++ Compiler (gcc-2.96). Para o ambiente paralelo foi usado o compilador `mpicc` – LAM 6.5.6/MPI 2 C++/ROMIO - University of Notre Dame.

4.1 Descrição do Algoritmo

Os documentos da coleção estão armazenados no diretório `/tmp/database/`, os quais são lidos pelo programa e associa um número a cada documento, o primeiro documento tem o valor 2, o segundo o valor 3, e assim por diante. O número 0 e 1 são reservados para indicar o diretório corrente e o diretório anterior, respectivamente.

O algoritmo para indexar o vocabulário de uma coleção de documentos é realizada em seis fases:

1. Inicialmente, uma lista com todas as *stopwords*, que estão em um arquivo chamado `stop.word`, é criada.

   A lista possui 1489 *stopwords* (palavras em inglês e português), a Tabela 4.1 mostra os números relacionados a cada idioma.

Tabela 4.1: Números de *stopwords*. * A letra 'c' não foi incluída, devido a linguagem C.

Inglês	400
Português	388
Sílabas c/ duas letras	676
Alfabeto	25*
Total	1489

2. A partir da lista de *stopwords* cria-se um autômato finito determinístico (AFD).

3. O AFD tem o objetivo de analizar lexicograficamente todos os caracteres que se encontram no texto, separando automaticamente todas as palavras que não são *stopwords* e substituindo os caracteres acentuados por aqueles sem acento (`ã` é substituido por `a`). Nesta fase também são eliminados os sinais de pontuação. A saída é gravada em arquivo, `/tmp/out.txt`, no formato:

   ```
   termo:processador:numero_do_documento
   ```

4. Após o pré-processamento do texto, uma nova lista com os termos retirados do documento é criada.

5. Nesta fase os termos retirados da coleção de documentos são ordenados lexicograficamente, os termos são ordenados por documento, ou seja, primeiramente estão ordenados todos os termos do documento 2, em seguida os termos do documento 3, e assim sucessivamente. Veja um exemplo:

   ```
   abrir:0:2
   abrir:0:2
   casa:0:2
   coelho:0:2
   coelho:0:2
   coelho:0:2
   violao:0:2
   abrir:0:3
   bahia:0:3
   bahia:0:3
   ```

```
violao:0:3
violao:0:3
violao:0:3
violao:0:3
violao:0:3
```

O algoritmo de ordenção utilizado foi o *quicksort*. A partir de então, é possível eliminar os termos repetidos percorrendo a lista uma única vez. A cada termo eliminado um contador é incrementado para que se possa saber quantas vezes determinado termo aparece no documento, ao final temos a seguinte saída.

```
termo:processador:documento:numero_de_ocorrencias

abrir:0:2:2
casa:0:2:1
coelho:0:2:3
violao:0:2:1
abrir:0:3:1
bahia:0:3:2
violao:0:3:5
```

6. Agora que temos todas as informações gravadas em arquivo, `sort.txt`, o arquivo invertido pode ser montado.

   A estrutura de dados utilizada para guardar as informações do arquivo invertido foi uma lista de adjacência encadeada. A lista foi criada utilizando alocação dinâmica de memória, por meio de ponteiros[1], a principal vantagem deste tipo de implementação é que novos itens podem ser inseridos e retirados sem haver a necessidade de deslocar os itens da lista. Esta estrutura foi adotada por ser de fácil implementação e porque o tamanho da lista não precisa ser definida a prior, a cada item inserido é alocado um espaço de memória dinamicamente. Uma desvantagem, destacada por [Ziv99], é a utilização de memória extra para armazenar os apontadores para o próximo item da lista.

   A cada novo termo inserido na lista ele é comparado item-a-item com os

[1] Ponteiro em C é uma variável que contém um endereço de memória de uma outra variável.

termos já presentes, se o termo está presente na lista as informações (processador, documento e número de ocorrências) são armazenadas na lista de adjacência, caso contrário elas são inseridas ao final da lista juntamente com o novo termo, veja a Figura 4.1.

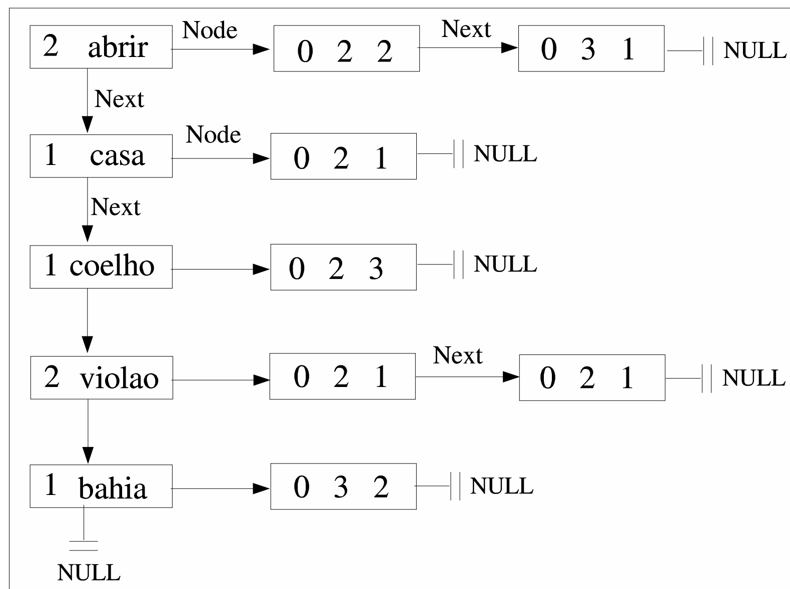

Figura 4.1: Modelo da Lista de Adjacência Encadeada

Ao término desta fase temos o arquivo invertido pronto para ser usado.

Veja no Apêndice B, página 53, a descrição das principais funções usadas pelo programa.

Capítulo 5

Algoritmo em Paralelo

Ao paralelizar a indexação dos termos de uma coleção de documentos a primeira coisa que deve-se pensar é como distribuir, ou pariticionar, a coleção entre os vários computadores. Há duas formas de se fazer isso. A primeira é o particionamento lógico e a segunda é o particionamento físico.

Partitionamento Lógico do Documento

Neste caso, o particionamento dos dados é feito logicamente, usando essencialmente o mesmo princípio básico do arquivo invertido seqüencial, onde a coleção é divida em blocos (arquivos, páginas da *Web*, tipos de documento, etc), os blocos podem ser de tamanho fixo ou variado. O arquivo invertido é estendido para dar a cada processador em paralelo acesso direto à porção do índice relacionado a subcoleção de documentos do processador. Cada entrada do termo no dicionário é estendido para incluir P ponteiros a lista invertida correspondente, onde o $j-th$ ponteiro indexa o bloco de documentos associado a lista invertida com a subcoleção do $j-th$ processador. Isso é mostrado na Figura 5.1, onde a entrada do dicionário para o termo i contém quatro ponteiros para a lista invertida de termos, um para cada processador paralelo ($P = 4$).

Particionamento Físico

Nesta segunda abordagem, os documentos são fisicamente particionados, em locais distindos, uma subcoleção para cada processador. Cada subcoleção possui seu próprio arquivo invertido. Quando uma consulta é submetida ao sistema, um

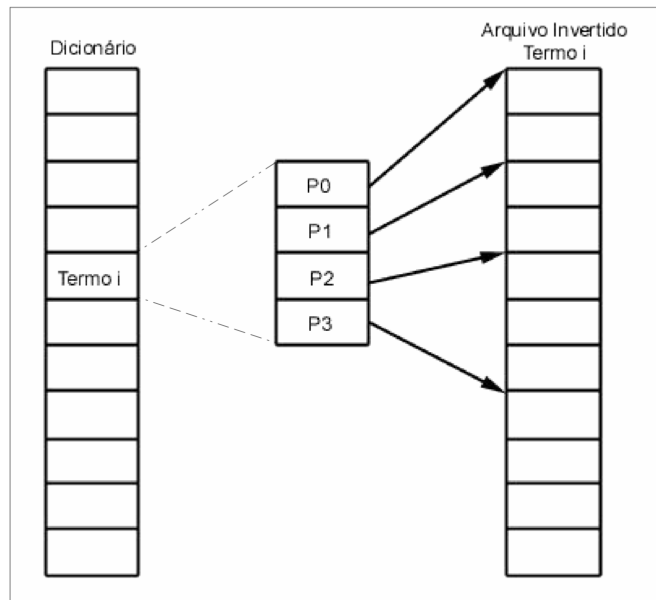

Figura 5.1: Entrada do dicionário estendido para um documento particionado.

agente distribui a consulta para todos os processadores paralelos. Cada processador avalia a pesquisa do termo baseado em sua subcoleção, o resultado é retornado ao agente. De posse de todos os resultados o agente junta as repostas para retornar ao usuário.

No presente trabalho foi adotado o particionamento físico, sendo que cada máquina possuia parte da coleção inicial e ao final do processamento cada processador tinha a lista invertida parcial da coleção.

5.1 Descrição do Algoritmo

O algoritmo proposto por [SZ97] é composto de três fases:

Arquivos Invertidos Locais: neste passo cada processador constrói o Arquivo Invertido para seu texto local.

Vocabulário Global: neste passo, o vocabulário global e a porção do Arquivo Invertido global a ser armazenada em cada processador, são determinados.

Distribuição Global do Arquivo Invertido: neste passo, porções dos Arquivos Invertidos locais são trocados para distribuir o arquivo invertido global, como determinado no passo 2.

Segue uma descrição suscinta das fases do algoritmo.

Arquivos Invertidos Locais

Neste primeiro passo, cada processador lê a parte de sua coleção em seu disco local e constrói o correspondente Arquivo Invertido. Não há necessidade de comunicação entre os processadores e esta fase é feita em paralelo nos p processadores.

A construção local do Arquivo Invertido consiste basicamente em se extrair o vocabulário, juntamente com o número de ocorrência de cada termo no documento e ordená-lo.

Vocabulário Global

Ao término do passo 1, cada processador possui o vocabulário local e o tamanho da lista de ocorrências (na coleção local) para cada termo do documento. Os processadores entram então em um processo de intercalação dos vocabulários locais, de forma a determinar o vocabulário global (juntamente com as listas de ocorrências de cada palavra). Para isto os processadores são então emparelhados, conforme mostra na Figura 5.2.

Na primeira etapa, os processadores são agrupados aos pares. O processador 0 é emparelhado com o 1, o processador 2 com o 3 e assim por diante. Assim o processador de maior índice transfere seu vocabulário e listas de ocorrências para o processador de menor índice, que intercala os vocabulários e atualiza as listas de ocorrências. Ao fim desta etapa, os processadores pares concentram todo o vocabulário do texto global.

Em uma segunda etapa, este agrupamento é repetido utilizando apenas os processadores de índice par. Após o fim da segunda etapa são agrupados os processadores de índice múltiplo de quatro, que no momento armazenam todo o vocabulário global.

Este agrupamento é aplicado recursivamente até que todo o vocabulário seja armazenado no processador de índice 0.

Segundo [SZ97] o processador 0 tem toda a informação necessária para determinação dos p intervalos iguais da Lista Invertida global, que ficarão armazenadas nos p processadores. Esta informação é então enviada a todos os p processadores.

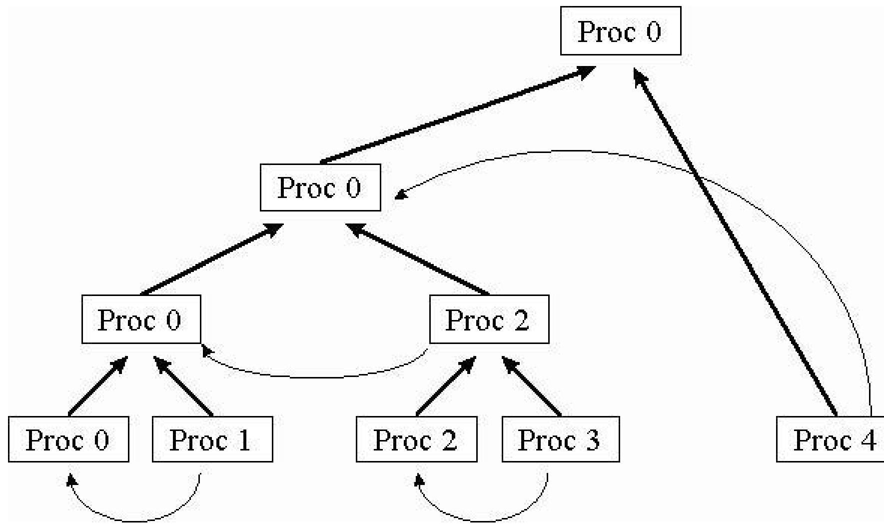

Figura 5.2: Construção do vocabulário global para cinco processadores.

Assim cada processador sabe qual o seu intervalo e qual o intervalo pertencente a cada um dos outros $p - 1$ processadores, sabendo então para onde enviar cada porção da seu arquivo Invertido local.

Distribuição Global do Arquivo Invertido

Como cada processador sabe que parte de seu Arquivo Invertido local deve ser transferido para que processador, uma seqüência de transferências deve ser utilizada de forma a realizar o menor número de etapas possível, explorando assim o máximo paralelismo de comunicação. Esta seqüência é pré calculada em cada processador. Como todos os processadores computam esta seqüência sobre uma mesma base de dados (os p processadores), a seqüência final obtida é a mesma para todos os processadores. Desta forma não são necessárias mensagens de sincronização.

A seqüência permite que em todo momento, cada processador esteja sendo emparelhado com um outro processador. Quando dois processadores são emparelhados eles trocam as apropriadas porções de seus índices. O mecanismo de trocas é dividido em etapas. Na primeira etapa, é garantido que todo processador seja emparelhado com seu sucessor e antecessor (considerando 0 e $p - 1$ como

vizinhos). Na segunda etapa, são emparelhados os processadores com distância dois e assim por diante. Desta forma $p/2$ etapas são necessárias. A Figura 5.3 ilustra o mecanismo de emparelhamento para sete processadores. Na etapa i, os processadores a distância i são emparelhados. Em cada etapa, são necessárias três fases para se emparelhar todos os vizinhos existentes. Agora, o arquivo invertido está pronto para ser pequisado.

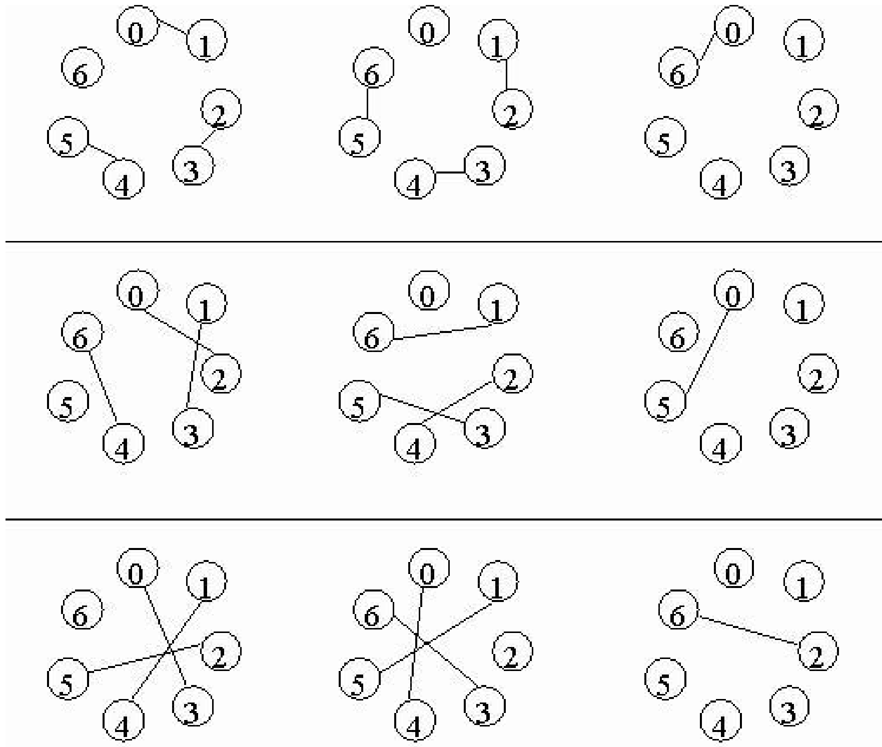

Figura 5.3: Processo de distribuição para sete processadores.

5.2 Implementação

O algoritmo descrito na Seção 5 não foi totalmente implementado. As fases 2 e 3 faltam ser implementadas. A maior parte do tempo do projeto foi usado no estudo de algoritmos, modelos de indexação e na implementação das operações básicas sobre um documento (análise léxica, eliminação de *stopwords*, etc). A análise léxica, eliminação das *stopwords* e a fase 1 do algoritmo em paralelo foi implementado com sucesso, sendo que cada processador constrói o Arquivo Invertido local para sua coleção de documentos.

A primeira fase do algoritmo paralelo foi implementada usando a linguagem C, o processo de sincronização entre os processadores foi realizado através do ambiente LAM/MPI. Para ativar o ambiente LAM/MPI é necessário seguir os passos descrito logo abaixo..

Ativando o Ambiente MPI

Para ativar os processadores das máquinas numa rede é necessário editar um arquivo, com a extensão `.def`, com os respectivos `IPs` das máquinas, por exemplo, o arquivo `hostsmpi.def` possui as seguintes linhas:

```
192.168.102.50
192.168.102.49
192.168.102.48
192.168.102.47
192.168.102.46
192.168.102.45
192.168.102.44
192.168.102.43
192.168.102.42
192.168.102.41
```

Caso a rede esteja habilitada com o login via `SSH`[1] será necessário gerar um par de chaves `DSA`.
Gerando um par de chaves DSA:

[1]SSH cliente é um programa para conectar em uma máquina remota e para a execução de comandos na máquina remota, numa rede com conexão criptografada.

```
[alisson@l2m50 mpi]$ ssh-keygen -d dsa
[alisson@l2m50 mpi]$ cd .ssh
[alisson@l2m50 mpi]$ cat id_dsa  id_dsa.pub  >  authorized_keys
```

Setando a variável de ambiente LAMRSH

```
[alisson@l2m50 mpi]$ setenv LAMRSH "ssh -x"
```

Reconhecendo os *hosts* da rede.

```
[alisson@l2m50 mpi]$ recon -av hostsmpifull.def
recon:  - testing n0 (192.168.102.50)
recon:  - testing n1 (192.168.102.49)
recon:  - testing n2 (192.168.102.48)
recon:  - testing n3 (192.168.102.47)
recon:  - testing n4 (192.168.102.46)
recon:  - testing n5 (192.168.102.45)
recon:  - testing n6 (192.168.102.44)
recon:  - testing n7 (192.168.102.43)
recon:  - testing n8 (192.168.102.42)
recon:  - testing n9 (192.168.102.41)

---------------------------------------------------
Woo hoo!

recon has completed successfully.  This means that you will
most likely be able to boot LAM successfully with the "lam-
boot"command (but this is not a guarantee).  See the lam-
boot(1) manual page for more information on the lamboot
command.

If you have problems booting LAM (with lamboot) even though
recon worked successfully, enable the -d"option to lam-
boot to examine each step of lamboot and see what fails.
Most situations where recon succeeds and lamboot fails have
to do with the hboot(1) command (that lamboot invokes on
each host in the hostfile).
```

```
--------------------------------------------------
```

Inicializando o ambiente LAM/MPI

```
[alisson@l2m50 mpi]$ lamboot -v hostsmpifull.def

LAM 6.5.6/MPI 2 C++/ROMIO - University of Notre Dame

Executing hboot on n0 (192.168.102.50 - 1 CPU)...
Executing hboot on n1 (192.168.102.49 - 1 CPU)...
Executing hboot on n2 (192.168.102.48 - 1 CPU)...
Executing hboot on n3 (192.168.102.47 - 1 CPU)...
Executing hboot on n4 (192.168.102.46 - 1 CPU)...
Executing hboot on n5 (192.168.102.45 - 1 CPU)...
Executing hboot on n6 (192.168.102.44 - 1 CPU)...
Executing hboot on n7 (192.168.102.43 - 1 CPU)...
Executing hboot on n8 (192.168.102.42 - 1 CPU)...
Executing hboot on n9 (192.168.102.41 - 1 CPU)...
topology done
[alisson@l2m50 mpi]$
```

Agora, o ambiente está pronto para executar o programa em paralelo.

```
[alisson@l2m50 mpi]$ mpirun N a.out
```

Para compilar o programa basta executar: **mpicc indexar.c**, o executável será um arquivo chamado `a.out`. Após a execução do algoritmo em paralelo é necessário que o ambiente LAM/MPI seja retirado do 'ar', para isso use o comando: **lamhalt**.

5.3 Consulta em Paralelo

Considerado que o Arquivo Invertido global está distribuído entre todas as máquina cada consulta submetida pode ser gerenciada por um agente que define qual máquina realizará consulta sobre o Arquivo Invertido, a resposta pode ser retornada diretamente ao usuário. O agente gerenciará as máquinas afim de repassar a consulta àquelas que estão em menor atividade.

Cada máquina será responsável pelo *ranking* da consulta de acordo com o

modelo adotado. Avaliar o grau de relevância de um determinado documento referente a uma consulta é uma tarefa não-trivial.

Um dos novos sistemas de busca que têm ganhado grande destaque é o Google [goo]. Segunda a empresa, seu mecanismo de busca calcula os resultados tomando por base um equação de 500 milhões de variáveis e mais de dois bilhões de termos. Em seus cálculos é levando em consideração a popularidade dos *links*. A posição de cada documento irá depender, entre outros fatores, do número de documentos que se ligam a ele e também da importância destes documentos. Ou seja a importância de um documento é derivada de sua popularidade e da popularidade dos documentos que apontam para ele [uni].

Nesta fase inicial, podemos considerar o documento mais relevante aquele que possui um maior número de ocorrências dos termos consultados, apoiando-se no modelo Booleano.

Capítulo 6

Conclusão

A computação paralela possui um grande potencial para lidar com grandes coleções de documentos. Utilizando um modelo apropriado de comunicação entre os processadores, pode-se obter ganhos bem elevados do poder de processamento em comparação aos sistemas seqüenciais.

Os resultados obtidos até aqui, inicilamente, são animadores. Um maior número de documentos pôde ser indexado, ressaltando que o resultado final foram Arquivos Invertidos locais sem comunicação entre os processadores, apenas sincronização.

O ideal é que o trabalho possa ter continuidade no desenvolvimento do algoritmo em trabalhos futuros. Dessa forma ampliaremos as possibilidades do desenvolvimento de sistemas de indexação e busca em base de textos, no intuito de criar tecnologia própria para um Sistema de Recuperação de Informação dentro da Universidade Federal de Lavras.

Trabalhos Futuros

Para a continuidade do trabalho sugere-se o desenvolvimento dos tópicos abaixo:

- Usar uma estrutura de dados mais eficiente para armazenar as informações do arquivo invertido, tal como, árvore-B;
- Implementar a fase 2 e 3 do algoritmo proposto em [SZ97].
- Fazer uma análise de complexidade do algoritmo;
- Medir o tempo de execução da geração do arquivo invertido *versus* o tamanho do texto de entrada;

- Uma análise quantitativa do rendimento do algoritmo, ou seja, até que ponto é viável a paralelização, atingindo o *speedup* máximo.
- Trabalhar com busca em Arquivos Invertidos compactados.
- Trabalhar com diferentes tipos de documentos, principalmente `.html`.

Referências Bibliográficas

[ALB96] J. Squyres A. Lumsdaine and B. Barrett. **LAM/MPI Parallel Computing**. Laboratory for Scientific Computing (LSC), Bloomington, Indiana University, 1996.

[Bar97] Ramurti Alencar Barbosa. **Recuperação de Informação em Sistemas Distribuídos**, 1997.

[BCCG95] Forbes J. Burkowski, G. Cormack, C. Clarke, and R. Good. **A global search architecture**, 1995.

[Beg94] A. Beguelin. **PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel computing**. The MIT Press, 1994.

[Bur90] Forbes J. Burkowski. **Retrieval performance of a distributed text database utilizing a parallel processor document server**, 1990.

[BYN97] R. Baeza-Yates and G. Navarro. **Block-addressing indices for approximate text retrieval.**, 1997.

[BYRN99] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. **Modern Information Retrieval**. ACM Press Book and Addison-Wesley, 1999.

[Cas99] Kalinka Regina Lucas Jaquie Castelo. **Extensão da Ferramenta de Apoio a Programação Paralela (F.A.P.P.) para ambientes paralelos virtuais**. Master's thesis, Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo, 1999.

[CC95] C. Clarke and G. Cormack. **Dynamic Inverted Indexes for a Distributed Full-Text Retrieval System**, 1995.

[Fly72] M. J. Flynn. **Some computer organizations and their effectiveness.** *IEEE Transactions on Computers*, 21(9):948–960, 1972.

[Fos95] Ian Foster. **Designing and Building Parallel Programs**, 1995.

[Fra92] Willian B. Frakes. **Information Retrieval - Data Structures & Algorithms**. Prentice Hall, 1992.

[goo] **http://www,google.com**.

[Goo77] S. E. Goodman. **Introduction to the Design and Analysis of Algorithms**. McGraw-Hill, 1977.

[IHWB99] Alistair Moffat Ian H. Witten and Timothy C. Bell. **Managing Gigabytes Compressing and Indexing Documents and Images**. Morgan Kaufmann Publishing,, 1999.

[JK97] N. Ziviani. J.P. Kitajima, B. Ribeiro. Parallel generation of inverted lists on a network of workstations., 1997.

[JO95] Byeong-Soo Jeong and Edward Omiecinski. **Inverted File Partitioning Schemes in Multiple Disk Systems**. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):142–153, 1995.

[Les] Michel Lesk. **The Seven Ages Of Information Retrieval**.

[MAZ97] G. Navarro M. Araújo and N. Ziviane. **Large text searching allowing erros.** *WSP'97*, pages 2–20, 1997.

[Mic82] A. Michard. **Graphical presentation of Boolean expressions in a database query language**, 1982.

[Moo51] Calvin N. Mooers. **Zatocoding Applied to Mechanical Organization of Knowledge**. American Documentation, USA, 1951.

[MS96] Steve Otto. et al Marc Snir. **MPI: The Complete Reference**. Massachusetts Institute of Technology, 1996.

[MZ] Alistair Moffat and Justin Zobel. **Information Retrieval Systems for Large Document Collections**.

[Qui94] Michael J. Quinn. **Parallel Computing Theory and Practice**. McGraw-Hill, 1994.

[SZ97] Berthier A. Ribeiro Neto; João Paulo Kitajima Gonzalo Navarro Cláudio R. G. Sant'Ana and Nivio Ziviani. **Parallel Generation of Inverted Files for Distributed Text Collections**, 1997.

[Tan97] Andrew S. Tanenbaum. **Organização Estruturada de Computadores**. Prentice/Hall do Brasil, 1997.

[TGM93a] Anthony Tomasic and Hector Garcia-Molina. **Performance of Inverted Indices in Distributed Text Document Retrieval Systems**. In *PDIS*, pages 8–17, 1993.

[TGM93b] Anthony Tomasic and Hector Garcia-Molina. **Performance of inverted indices in shared-nothing distributed text document information retrieval systems**, 1993.

[uni] **Revista Eletrônica Unicamp: Google, Microsoft e outros mais**. http://www.revista.unicamp.br/infotec/internet/internet13-1.html.

[YS93] Degi Young and Ben Shneiderman. **A graphical filter/flow model for boolean queries: An implementation and experiment.** *Jounal of the American Society for Information Science*, pages 327–339, 1993.

[Ziv99] Nivio Ziviane. **Projeto de Algoritmos**. II. Pioneira Informática, São Paulo, 4 edition, 1999.

Apêndice A

Lei de Heaps

Para prever o crescimento do tamanho de um vocabulário num texto usa-se a lei de Heaps. Segundo [BYRN99], esta é uma lei muito precisa, no qual um texto com n palavras possui uma tamanho $V = Kn^\beta = \mathrm{O}(n^\beta)$, onde K e β depende do texto em particular, a Figura A.1 ilustra como o tamanho do vocabulário de um texto varia. K normalmente está entre 10 e 100, e β é um valor positivo menor que um. Alguns experimentos, [MAZ97] [BYN97] na coleção TREC-2 demonstraram que o valor mais comum para β está entre $0,4$ e $0,6$. Conseqüentemente, o vocabulário do texto cresce sublinearmente com o tamanho do texto.

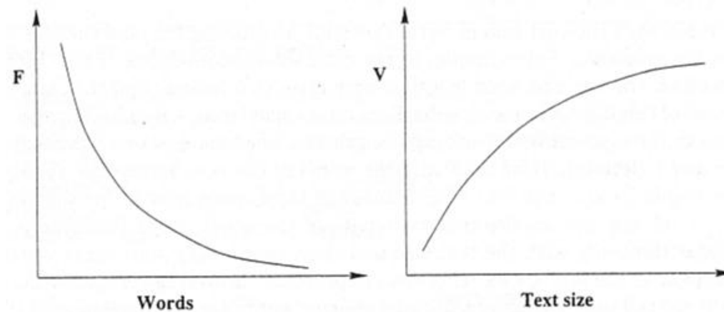

Figura A.1: A esquerda, distribuição da freqüência das palavras. A direita, o tamanho do vocabulário. [BYRN99]

Apêndice B

Funções

B.1 Arquivo `fileinverted.c`

`ArchiveInsert( ArchiveInverted file, char *string, int processor, int idDoc, int occur )`

Propósito: Inserir um termo na estrutura de dados juntamente com o número do processador, o documento em que o termo ocorre e quantas vezes o termo ocorre. Os termos são inseridos na ordem lexicográfica.
Retorno: A função retorna um tipo **int**, zero se ocorrer algum erro e um se o termo foi inserido com sucesso.

`ArchivePrint( ArchiveInverted file )`

Propósito: Imprimir todos os termos do arquivo invertido.

`ArchiveMount( ArchiveInverted file, char *filename )`

Propósito: Montar o arquivo invertido em memória com base nos dados gravado no arquivo apontado pelo ponteiro *filename*.
Retorno: o número de termos distintos que o arquivo invertido possui.

A estrutura do Arquivo Invertido é composta por duas estruturas.

A primeira é um lista encadeada com as informações relativa a um determinado termo,

- proc número do processador que o documento está armazenado;

- idDoc identificador do documento na coleção local;

- occur número de vezes que um termo ocorre no documento;

- next por último, um ponteiro para uma próxima estrutura deste tipo.

A segunda estrutura é uma outra lista encadeada dos termos do arquivo invertido,

- size o valor total que cada termo aparece em toda coleção, isso é útil para que os vários processadores saibam quantos termos vão receber de cada processador;

- string o termo indexado;

- node um ponteiro para a estrutura *Posting*;

- next um ponteiro que aponta para o próximo termo (string) da lista.

```
typedef struct Posting
{
    unsigned int proc;
    unsigned int idDoc;
    unsigned int occur;
    struct Posting *next;
}
Posting;

typedef struct _ArchiveInvertedStruct
{
    int size;
    char *string;
    struct Posting *node;
    ArchiveInverted next;
}ArchiveInvertedStruct;
```

A Figura B.1 ilustra a estrutura:

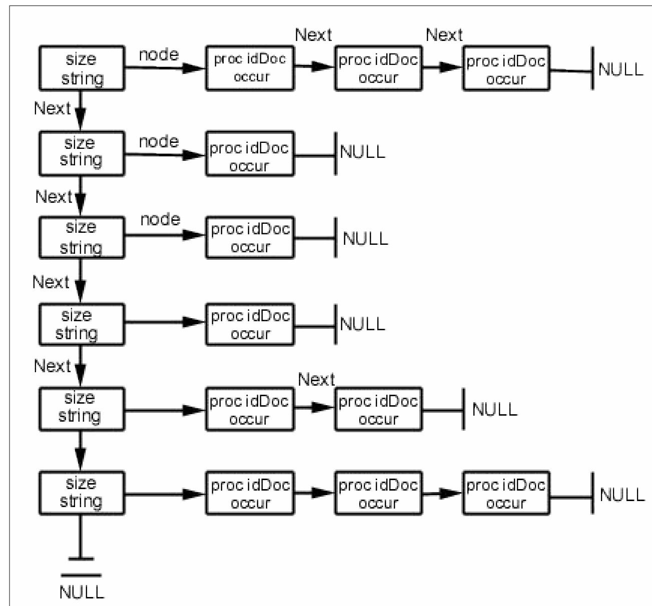

Figura B.1: Estrutura do Arquivo Invertido

B.2 Arquivo `stop.c`

As duas principais funções são:

`BuildDFA( StrList words )`

Propósito: Criar um Autômato Finito Deterministico (AFD) para reconhecer as *stopwords* nos documentos.
Retorno: Devolve um máquina finita de estados, com base no AFD.

`GetTerm( FILE *stream, DFA machine, int size, char *output )`

Propósito: Separar os termos do *stream* de entrada, filtrando as *stopwords*. A filtragem é feita com base na máquina finita de estados que foi criada pela função `BuildDFA(...)`.
Retorno: Cada termo reconhecido é retornado e gravado em disco, no arquivo `out.txt`. Isso é feito até que todo o documento seja lido.

B.3 Arquivo `strlist.c`

As principais funções são:

`ExpandArray( StrList list )`
: **Propósito:** Aumentar a lista de *strings* a cada vez que a lista atingir o tamanho máximo, evitando a cada inserção de uma *string* fazer a alocação de memória. Obs: o incremento é de 128 itens.
Retorno: retorna o valor 1 se a lista foi realocada com sucesso e o valor 0 caso contrário.

`QSort( char **string, int lb, int ub )`
: **Propósito:** Ordenar um lista de *strings* utilizando o algoritmo *quicksort*.

`StrListAppendFile( StrList list, char *filename )`
: **propósito:** Criar uma lista de *string* a partir de um arquivo de entrada.

`StrListCreate( )`
: **Propósito:** Alocar e inicializar uma lista de *strings*.
Retorno: retorna um ponteiro para a estrutura criada, apontando para o primeiro elemento da lista.

`StrListUniqueOccur( list )`
: **Propósito:** Retirar todas as *strings* duplicadas que estão na lista.