PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS
Programa de Pós-graduação em Engenharia Elétrica

Ana Lúcia Martins de Jesus

PROPOSTA E AVALIAÇÃO DE DESEMPENHO DE UMA IMPLEMENTAÇÃO PARALELA DA EXTRAÇÃO DE ATRIBUTOS MFCC

Belo Horizonte
2015

Ana Lúcia Martins de Jesus

PROPOSTA E AVALIAÇÃO DE DESEMPENHO DE UMA IMPLEMENTAÇÃO PARALELA DA EXTRAÇÃO DE ATRIBUTOS MFCC

Dissertação apresentada ao Programa de Pós-graduação em Engenharia Elétrica da Pontifícia Universidade Católica de Minas Gerais, como requisito parcial para obtenção do título de Mestre em Engenharia Elétrica.

Orientador: Prof. Dr. Luís Fabrício Wanderley Góes
Coorientadora: Profa. Dra. Flávia Magalhães Freitas Ferreira

Belo Horizonte
2015

FICHA CATALOGRÁFICA
Elaborada pela Biblioteca da Pontifícia Universidade Católica de Minas Gerais

Jesus, Ana Lúcia Martins de
J58p Proposta e avaliação de desempenho de uma implementação paralela da extração de atributos MFCC / Ana Lúcia Martins de Jesus. Belo Horizonte, 2015.
66 f. : il.

Orientador: Luís Fabrício Wanderley Góes
Coorientadora: Flávia Magalhães Freitas Ferreira
Dissertação (Mestrado) – Pontifícia Universidade Católica de Minas Gerais. Programa de Pós-Graduação em Engenharia Elétrica

1. Programação paralela (Computação). 2. Reconhecimento automático da voz. 3. Processamento de sinais - Técnicas digitais. 4. Sistemas de reconhecimento de padrões. 5. OpenMP (Application program interface). I. Góes, Luís Fabrício Wanderley. II. Ferreira, Flávia Magalhães Freitas. III. Pontifícia Universidade Católica de Minas Gerais. Programa de Pós-Graduação em Engenharia Elétrica. IV. Título.

CDU: 621.391

Ficha catalográfica elaborada por Fernanda Paim Brito - CRB 6/2999

Ana Lúcia Martins de Jesus

PROPOSTA E AVALIAÇÃO DE DESEMPENHO DE UMA IMPLEMENTAÇÃO PARALELA DA EXTRAÇÃO DE ATRIBUTOS MFCC

Dissertação apresentada ao Programa de Pós-graduação em Engenharia Elétrica da Pontifícia Universidade Católica de Minas Gerais, como requisito parcial para obtenção do Título de Mestre em Engenharia Elétricaem Programa de Pós-graduação em Engenharia Elétrica.

Prof. Dr. Luís Fabrício Wanderley Góes (Orientador) - PUC Minas

---

Prof. Dr. Luiz Eduardo da Silva Ramos - Rutgers University

Prof. Dr. Carlos Augusto Paiva da Silva Martins - PUC Minas

Belo Horizonte, 1 de Julho de 2015.

Dedicatória: Dedico a meus pais, irmãos, orientadores e aos amigos que tanto apoiaram, principalmente em momentos tão difíceis..

AGRADECIMENTOS

Agradeço a Deus pela oportunidade de trilhar esse caminho tão difícil e pela força ao concluir o trabalho.

Sou muito grata à minha família pela compreensão em virtude de minhas ausências, e pelo total apoio.

A meus orientadores Luís Fabrício e coorientadora Flávia Magalhães, que acreditaram neste trabalho, pela compreensão e pela motivação.

Agradeço ao professor Dorirley, pela indicação.

Agradeço a Willian Antônio, Willian Gomes e Harison Silva pela boa vontade e disposição em ajudar, principalmente ao Willian Antônio, autor da implementação sequencial do ASR objeto de estudo neste trabalho.

Ao professor Carlos Augusto, por estar sempre disposto a ajudar.

Aos colegas de mestrado pelo companheirismo e solidariedade.

A Fábio, Aguinaldo e Caique pela amizade e boa convivência.

Aos colegas de trabalho no setor de informática do CRMMG, pela compreensão, pela boa convivência e principalmente pela força.

Ao pessoal que trabalha na secretaria do PPGEE, Isabel Siqueira, Isabel Novaes e Eliza pelo profissionalismo e simpatia.

A CAPES, pelo incentivo.

Agradeço a todos aqueles que se envolveram direta e indiretamente na realização desta pesquisa.

Tem dias que a gente se sente
Como quem partiu ou morreu
A gente estancou de repente
Ou foi o mundo então que cresceu
A gente quer ter voz ativa
No nosso destino mandar
Mas eis que chega a roda-viva
E carrega o destino pra lá

(BUARQUE, 1967)

RESUMO

A eficiência dos sistemas de processamento de voz depende da qualidade da informação extraída das amostras de áudio. Uma técnica muito utilizada para adquirir esta informação é a extração de atributos Mel Frequency Cepstral Coefficents (MFCC). Porém, esta técnica demanda um alto custo computacional e sua paralelização não é trivial, constada pelo baixo desempenho alcançado em trabalhos relacionados. Neste trabalho é proposto, implementado e avaliado o desempenho de uma versão paralela em OpenMP de um extrator de atributos acústicos baseado em coeficientes MFCC. Os resultados experimentais apresentam um *speedup* de até 2,6 em um máquina com múltiplos núcleos.

Palavras-chave: Reconhecimento de voz. ASR. Computação paralela. OpenMP. Palavras isoladas. MFCC. *Front-end*

ABSTRACT

The efficiency of Voice processing systems efficiency relies on the quality of the information extracted from audio samples. A technique widely used to acquire this information is the Mel Frequency Cepstral Coefficents (MFCC) attribute extraction. However, this technique requires a high computational cost and parallelization is not trivial, verified by the low performance achieved in related work. In this paper we proposed, implemented and evaluated the performance of a parallel version of of an acoustic attributes extractor based on MFCC coefficients. The experimental results show a *speedup* of up to 2.6 on a *multicore*.

Keywords: Voice Recognition. ASR. Parallel Computing. OpenMP. Isolated words. Coeficientes Mel Cepestrais. Extração de atributos.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE GRÁFICOS

LISTA DE SIGLAS

API – Application Program Interface
ASR – Automatic Speech Recognition System
DCT – Discrete Cosine Transform
FFT – Fast Fourier Transform
FPGA – Field Programmable Gate Array
GPU – Graphics Processing Unit
HTK – Hidden Markov Model Toolkit
MFCC – Mel-Frequency Cepestral Coefficients
MPI – Message Passing Interface
OpenMP – Open Multi-Processing
PAPI – Performance Application Programming Interface

SUMÁRIO

1 INTRODUÇÃO

Atributos MFCC (DAVIS; MERMELSTEIN, 1980) (*Mel Frequency Cepstral Coefficients*) são considerados os mais importantes atributos utilizados no pré-processamento do áudio para sistemas de processamento de voz em geral (BAHOURA; EZZAIDI, 2013). Tal representação caracteriza-se por possibilitar o mapeamento da percepção auditiva humana de frequências puras, fazendo a correspondência com a escala Hertz.

Os atributos MFCC surgiram devido a estudos da área da psicoacústica, que mostraram que a percepção auditiva humana aos tons puros (frequências puras) do sinal de voz não segue uma escala linear. Dessa forma, foi proposta uma escala que mapeia as frequências subjetivas dos tons puros, chamada de escala Mel, na escala Hertz (Hz) de frequência.

Para a determinação da relação entre a escala mel e a escala Hz, foi realizado um experimento onde foi definida a frequência de 1.000Hz, com 40dB SPL (*Sound Pressure Level*) acima do limiar de audição humana, correspondendo a 1.000mels. Os outros valores foram encontrados pedindo-se que ouvintes ajustassem a frequência física (em Hz) de tom em tom, até que a frequência percebida fosse duas vezes a frequência de referência. Em seguida, o mesmo experimento foi realizado considerando-se 10 vezes a frequência de referência, e assim por diante. Dessa forma, as frequências encontradas teriam valores de 2.000mels e 10.000mels, respectivamente. De forma semelhante, o processo foi realizado solicitando-se aos ouvintes que ajustassem a frequência para metade e para um décimo da frequência de referência, obtendo-se dessa forma as frequências de 500mels e 100mels, respectivamente. Com esse mapeamento, foi possível verificar que a relação entre a frequência em Hz e a frequência percebida é aproximadamente linear abaixo de 1.000Hz, mas apresenta um comportamento logarítmico acima desse valor.

Pelo fato de proporcionarem extração de informações do áudio de forma compacta, dentre outras características, atributos MFCC são amplamente utilizados na literatura. Assim como outros algoritmos populares de extração de informação de conteúdo de aúdio, a extração de atributos MFCC utiliza processamento de sinal sofisticado e requer grande poder computacional (MAKA; DZIURZANSKI, 2013). Para possibilitar a extração de recursos cada vez mais rápida mantendo a fidelidade dos resultados, diversas pesquisas têm proposto a aplicação de computação paralela na realização deste processo (BAHOURA; EZZAIDI, 2013; KOU et al., 2013; KOU; SHANG, 2014; MAJID; MIRZAEI; JAMALI, 2012; MAKA; DZIURZANSKI, 2013). Este trabalho utiliza computação paralela na implementação de um extrator de atributos baseado em MFCC, sendo implementado na linguagem C juntamente com a API (*Application Program Interface*) de programação paralela OpenMP(*Open Multi-Processing*).

1.1 Justificativa

O processo de extração de atributos acústicos envolve quantidade significativa de cálculos e processamento de dados. Torna-se necessário utilizar estratégias para diminuir o tempo de processamento, mantendo a taxa de reconhecimento acima de 90%. Uma solução para esse tipo de problema é a implementação paralela (YOU; LEE; SUNG, 2009; MAKA; DZIURZANSKI, 2013).

Dado o fato de a extração de atributos acústicos ser uma etapa comum a diversos tipos de sistemas de processamento de voz, esse estudo pode ser estendido também à melhoria de outros tipos de aplicações tais como indexação de som (MAKA; DZIURZANSKI, 2013), análise de sons da respiração (BAHOURA; EZZAIDI, 2013), reconhecimento de emoções na fala (DAN; MONICA, 2013) e detecção de atividades de voz para uso em sistema de reconhecimento de fala contínua (WANG; XU; LI, 2011), dentre outras.

1.2 Objetivo geral

Esta pesquisa tem por objetivos propor, implementar e avaliar o desempenho de uma versão paralela em OpenMP de um extrator de atributos acústicos baseado em coeficientes MFCC. Para validar o processo, o extrator será utilizado em um sistema de reconhecimento de palavras isoladas. A utilização de OpenMP é motivada pela portabilidade de código em diferentes sistemas operacionais, sendo a mudança de ambiente uma questão de recompilação (CHANDRA et al., 2001). Por ser uma API de programação paralela em processadores com múltiplos núcleos, é amplamente disponível em sistemas computacionais de relativo baixo custo, como computadores pessoais.

1.3 Estrutura da dissertação

O restante do trabalho está organizado da seguinte forma: No Capítulo 2 são abordados os conceitos de Extração de Atributos MFCC, Computação Paralela e sobre a API OpenMP. O Capítulo 3 apresenta os principais trabalhos correlatos na área de implementação paralela da extração de atributos em sistemas de processamento de voz. No Capítulo 4 são descritas as etapas de implementação do sistema objeto deste trabalho, utilizando computação paralela. O Capítulo 5 faz uma descrição do método de avaliação adotado na pesquisa. Os resultados experimentais são relatados no Capítulo 6 e por fim, o Capítulo 7 apresenta a conclusão obtida na realização do trabalho.

2 FUNDAMENTAÇÃO TEÓRICA

A seguir, são descritos os principais tópicos abordados neste trabalho: conceitos relacionados à extração de atributos no *Front-end* e programação paralela.

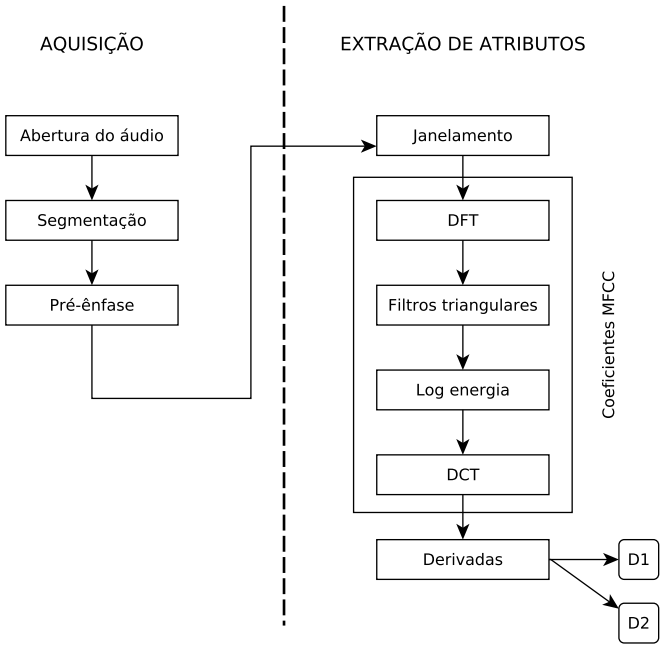
2.1 EXTRAÇÃO DE ATRIBUTOS MFCC

O sistema de reconhecimento de fala implementado neste trabalho utiliza MFCC na extração dos atributos acústicos e Modelo Oculto de Markov (*Hidden Markov Model* - HMM) para reconhecimento das palavras. Atributos MFCC são caracterizados por representarem o sinal de áudio em atributos compactos, que simulam a percepção através de audição humana. Além disso, atributos MFCC são os recursos mais utilizados na literatura, sendo muitas vezes utilizados como atributos padrão por muitas organizações de pesquisa (LAWSON et al., 2011). As seções a seguir descrevem o processo de aquisição e conversão das amostras de áudio em coeficientes MFCC.

2.1.1 Aquisição as amostras de áudio

A *Aquisição do Áudio* caracteriza a obtenção das amostras de áudio pelo extrator. A *Extração de Atributos* engloba a extração dos coeficientes MFCC propriamente ditos, como será descrito adiante.

Figura 1 – *Front-end*



Fonte: dados da pesquisa

Os passos para a Extração de Atributos são descritos a seguir e a visão geral do processo pode ser conferida na Figura 1

Abertura do áudio: é realizada através da leitura do arquivo ou da captação por meio de um microfone, procedendo ao carregamento das amostras para a memória. A abertura se dá tipicamente por algum meio elétrico ou mecânico, dentre os quais se destacam as classes analógica e digital.

Segmentação: Conforme apontado por (SANTOS, 1997), para a maioria dos sistemas de processamento de voz, são necessárias algumas suposições a respeito das características dos sinais de voz para que se proponha uma modelagem acústica. Uma das mais importantes é que o sinal pode ser considerado estacionário em um intervalo pequeno de tempo, o que, em outras palavras, quer dizer que suas características espectrais são constantes nesse intervalo. Diante disso, para que essa suposição possa ser satisfeita, é necessário segmentar o sinal de voz em blocos de curta duração (RABINER, 1989). Empiricamente foi constatado que um tamanho adequado para esses blocos está em torno de 20 a 40ms. A fim de preservar as características temporais do sinal de fala na modelagem acústica, é necessária uma sobreposição entre blocos consecutivos, sendo que a quantidade geralmente utilizada de amostras sobrepostas fica em torno de 25% a 75% do total das amostras do bloco (SANTOS, 1997). A título de exemplo, suponha um sinal divido em blocos de 20ms e com sobreposição de 50%. Nesse caso, um novo bloco deve ser tomado a cada 10ms e terá 50% da informação do bloco anterior. Essa divisão do áudio caracteriza a etapa de segmentação.

Pré-ênfase: Após a segmentação, submete-se o sinal à filtragem pré-ênfase, com objetivo de atenuar as baixas frequências sinal, de modo a compensar a atenuação das altas frequências do sinal de voz pelos lábios. A função de transferência do filtro de pré-ênfase mais utilizado é mostrada na Equação 1, onde o parâmetro α é chamado coeficiente de pré-ênfase, assumindo normalmente o valor $0,97$ (SANTOS, 1997).

$$H(z) = 1 - \alpha \cdot z^{-1} \tag{1}$$

A subseção a seguir apresenta os conceitos relacionados à Extração de Atributos propriamente dita.

2.1.2 Extração de atributos MFCC

Uma vez realizados os processos de abertura ou captação do áudio, segmentação em blocos e aplicação do filtro pré-ênfase, que caracterizam a etapa de aquisição, procede-se às etapas relativas à **Extração de Atributos**. Nessa etapa é realizado o janelamento, cálculo da DFT, cálculo dos filtros triangulares, da DCT e derivadas, conforme apresen-

tado a seguir.

Janelamento: Com o efeito da segmentação do sinal de voz em blocos realizada na etapa de aquisição, ocorre uma grande descontinuidade nas bordas de cada bloco, o que ocasiona um fenômeno espectral indesejável chamado fenômeno de Gibbs (SANTOS, 1997). Para contornar esse problema, aplica-se uma janela a cada bloco, sendo mais utilizada a janela de Hamming (HUANG; ACERO; HON, 2001). A janela de Hamming tem o papel de suavizar as bordas dos blocos, e é descrita pela Equação 2, em que n é a n-ésima amostra em um bloco e N é o número de amostras deste bloco.

$$Hamming(n) = 0.54 - 0.46cos(2\pi \cdot n \cdot N) \tag{2}$$

No processo de extração de atributos, são realizadas estimativas espectrais em cada bloco e então, essas estimativas passam por uma série de transformações, até que sejam obtidos os atributos acústicos correspondentes a cada um deles. Os atributos acústicos têm por objetivo oferecer ao sistema de processamento de voz, uma representação compacta e consistente do sinal, buscando evidenciar as características que discriminam os diferentes sons da fala, ao mesmo tempo em que elimina as informações menos relevantes. Os atributos utilizados neste trabalho são descritos na subseção a seguir.

2.1.2.1 Coeficientes mel cepestrais (MFCC)

Os coeficientes MFCC são baseados em uma escala que mapeia as frequências subjetivas dos tons puros, chamada de mel, na escala Hertz (Hz) de frequência. A Equação 3 apresenta uma aproximação do comportamento da frequência percebida B, em mel, em termos da frequência f, expressa em Hz.

$$B(f) = 1125 \ln(1 + \frac{f}{700}), \quad 0 \leq k < N \tag{3}$$

O cálculo dos coeficientes MFCC é composto pelo cálculo da DFT, dos filtros triangulares, da DCT, do log de energia e opcionalmente, pelo cálculo das derivadas, conforme descrito a seguir.

Cálculo da DFT: A primeira etapa do processo de geração dos coeficientes MFCC consiste em calcular a Transformada Discreta de Fourier (*Discrete Fourier Transform* – DFT) das amostras de um bloco de voz. Esta transformada é definida na Equação 4, onde N é o número de pontos da transformada, $s[n]$, a n-ésima amostra do sinal de entrada e S_a, o k-ésimo elemento do resultado da Transformada.

$$S_a[k] = \sum_{n=0}^{N-1} s[n] e^{-j2\pi nk/N}, \quad 0 \leq k < N \tag{4}$$

Filtros triangulares: Logo após o cálculo da DFT, é definido um banco de filtros triangulares e igualmente espaçados na escala mel, com resposta em frequência definida a partir da Equação 5.

$$H_m(k) = \begin{cases} 0, & k < f[m-1] \\ \frac{(k - f[m-1])}{f[m] - f[m-1]} & f[m-1] \leq k \leq f[m] \\ \frac{(f[m+1] - k)}{f[f+1] - f[m]} & f[m] \leq k \leq f[m+1] \\ 0, & k > f[m+1] \end{cases} \tag{5}$$

Na Equação 5, $f[m]$ representa a frequência central em Hz (dada em termos dos índices da transformada de Fourier) do m-ésimo filtro do banco de filtros, que é calculado a partir da Equação 6, onde f_l e f_h são, respectivamente, as frequências máxima e mínima do sinal considerado. N é o número de pontos da transformada de Fourier, M é o índice do filtro no banco de filtros e F_s é a frequência de amostragem do sinal.

$$f[m] = \left(\frac{N}{F_s}\right) B^{-1}\left(B(f_l) + m\frac{B(f_h) - B(f_l)}{M+1}\right) \tag{6}$$

Em seguida, a energia logarítmica de cada filtro é calculada de acordo com a Equação 7.

$$S[m] = \ln\left[\sum_{k=0}^{N-1} \|S_a[k]\|^2 H_m[k]\right] \tag{7}$$

DCT: Após a aplicação dos filtros triangulares, aplica-se a Transformada Discreta do Cosseno (DCT), definida na Equação 8, sobre os M valores de energia logarítmica dos filtros para se obter os coeficientes no domínio da frequência. Esse procedimento visa comprimir a informação espectral em coeficientes de baixa ordem (SANTOS, 1997), de forma que, no domínio DCT, os coeficientes MFCC de maior energia correspondam aos coeficientes de menores índices n. Além disso, a DCT causa o efeito benéfico, em termos de eficiência de representação do sinal, de produzir coeficientes MFCC praticamente descorrelatados (ALENCAR, 2005). Em ASR são utilizados valores para M que variam entre 24 e 40, e geralmente os vetores de atributos MFCC de cada bloco do sinal são formados

pelos primeiros 13 coeficientes ([k).

$$c_l(k) = \sum_{m=1}^{M} e(l,m) cos\left(k(m-0.5)\frac{\pi}{M}\right) \tag{8}$$

onde:

m = m-ésimo filtro do banco

l = índice do *frame*

k = índice de amostras

M = número de filtros

2.1.2.2 Cálculo da energia e derivadas

Log de energia: Na composição dos vetores de atributos acústicos obtidos com MFCC, é comum a adição da energia de cada bloco de voz. A energia é calculada como o logaritmo da energia do sinal, ou seja, para as amostras $s[n]$, com $1 \leq n \leq N$, onde N é o número de amostras, a energia pode ser obtida a partir da Equação 9.

$$E = \log \sum_{n=1}^{N} s[n]^2 \tag{9}$$

Derivadas: A taxa de acerto do sistema de reconhecimento automático de voz pode ser significativamente melhorada através da adição das derivadas no tempo do sinal aos vetores de atributos. Esses novos componentes dos vetores acústicos, também chamados de componentes dinâmicos, em contraste aos componentes estáticos obtidos como mostrado anteriormente, têm o objetivo de incorporar informação da variação temporal do sinal de voz aos vetores de atributos. Assim, é possível capturar os efeitos de coarticulação causados pela inércia do aparelho vocal humano (SANTOS, 1997). Os atributos dinâmicos mais utilizados são a derivada primeira, também conhecidos como coeficientes delta, e a derivada segunda, também chamados de coeficientes de aceleração. As Equações 10 e 11 apresentam o cálculo das referidas derivadas, em que Δ_i(n) representa a derivada primeira e D_i(n) a derivada segunda. $C_{i(+-K)}(n)$ é o n-ésimo coeficiente do vetor i e ϕ é um parâmetro de distância relativa usado no cálculo de ambos, geralmente igual a 2.

$$\Delta_i(n) = \frac{\sum_{k=1}^{\phi} k \cdot (C_{i+k}(n) - C_{i-k}(n))}{(2 \cdot \sum_{k=1}^{\phi} k^2)} \tag{10}$$

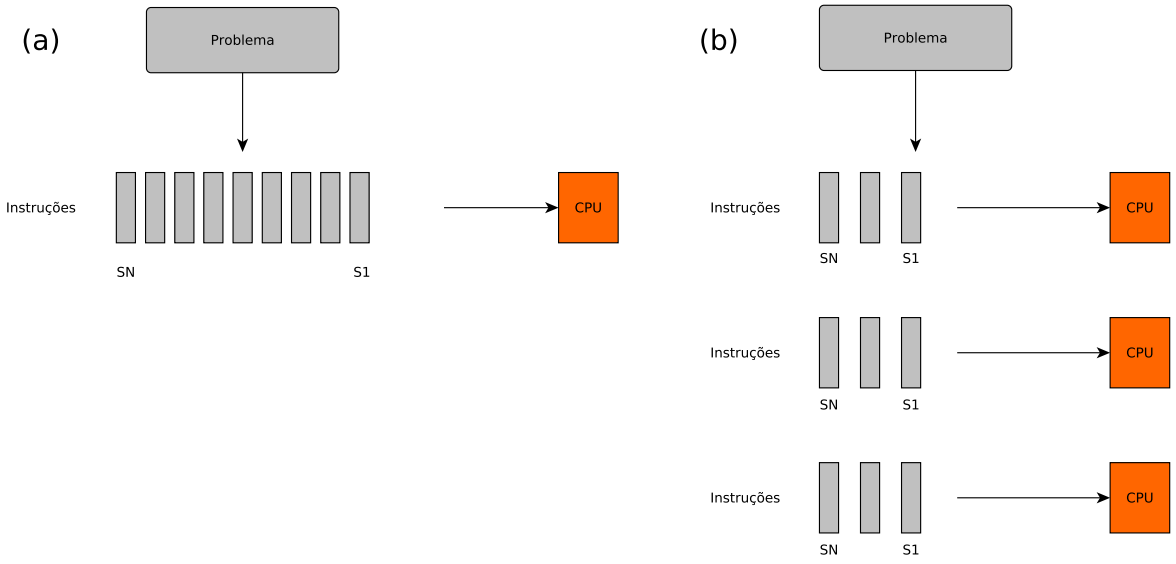
$$D_i(n) = \frac{\sum_{k=1}^{\phi} k \cdot (\Delta_{i+k}(n) - \Delta_{i-k}(n))}{(2 \cdot \sum_{k=1}^{\phi} k^2)} \quad (11)$$

O processamento dos atributos MFCC possibilita uma representação compacta do sinal de áudio analisado. Ao final da extração de atributos, cada bloco é representado por um vetor de coeficientes MFCC. No caso deste trabalho, foram utilizados vetores com 13 componentes, sendo 12 atributos MFCC e 1 coeficiente 0. As seções a seguir apontam os principais conceitos de computação paralela adotados neste trabalho.

2.2 PROGRAMAÇÃO PARALELA

Segundo (RAJASEKARAN; REIF, 2008), a computação paralela é definida como a execução de várias tarefas do mesmo tipo simultaneamente, tendo por principal objetivo reduzir o tempo total de processamento. Diferentemente da programação sequencial, cujas instruções são executadas em apenas um processador sequencialmente, conforme ilustrado na Figura 2(a), as tarefas que compõem o problema podem ter suas instruções, de S1 a SN, executadas simultaneamente nos processadores disponíveis, conforme apresentado na Figura 2(b). Na utilização de P processadores, existe potencial de reduzir o tempo de execução sequencial em um fator de até P vezes. Diante disso, esse modelo de programação é utilizado para solucionar problemas que requerem alto poder de processamento.

Figura 2 – (a)Programação sequencial (b) Programação paralela



Fonte: Dados da pesquisa

A implementação de um programa paralelo é fortemente influenciada pelo sistema de processamento paralelo utilizado (RAUBER; RUNGER, 2010), que compreende

os componentes de *hardware* e *software* disponíveis para o desenvolvimento da aplicação. Para os diversos tipos de *hardware* disponíveis, foram propostos vários modelos de programação com o objetivo de proporcionar o aproveitamento dos recursos de maneira mais eficiente possível. Desta maneira, aplicações paralelas podem ser implementadas utilizando-se linguagens de programação específicas, APIs ou bibliotecas em tempo de execução.

As seções a seguir apresentam conceitos relacionados a computação paralela. Na seção 2.2.1 são apresentados os modelos de arquiteturas. A seção 2.2.2 apresenta a API de programação paralela OpenMP juntamente com os conceitos utilizados neste trabalho. Por fim, a identificação, a classificação e a remoção de dependências em códigos paralelos são abordados nas seções 2.2.3, 2.2.4 e 2.2.5, respectivamente.

2.2.1 Arquiteturas paralelas

A possibilidade de execução de aplicações paralelas depende fortemente da arquitetura da plataforma utilizada (RAUBER; RUNGER, 2010), que determina como as tarefas de um programa podem ser mapeadas para os recursos disponíveis. Existem diversos tipos de arquitetura que podem ser classificados por modelos. O Modelo de Flynn, descrito em (FLYNN, 1972), é o mais aceito e distingue 4 modelos de arquitetura:

a) ***Single-Instruction, Single-Data* (SISD):** Nesse modelo existe apenas uma unidade de processamento. A cada etapa, a unidade de processamento lê os dados correspondentes e executa a instrução, armazenando o resultado. Este modelo é a convencional representação do computador sequencial proposto no modelo de Von Neumann (RAUBER; RUNGER, 2010);

b) ***Multiple-Instruction, Single-Data* (MISD):** Este modelo caracteriza a existência de múltiplos elementos de processamento, sendo que cada um tem seu local privado de memória. A cada etapa, são executadas diferentes instruções em paralelo e o mesmo elemento de dados é carregado da memória. É um modelo de execução restritivo e não possui representação comercial;

c) ***Single-Instruction, Multiple-Data (SIMD*):** Neste modelo existem várias unidades de processamento e cada uma delas tem acesso privado a uma memória de dados, que pode ser compartilhada ou distribuída. No entanto, há apenas uma memória a partir da qual um processador de controle especial busca e despacha instruções. A cada etapa, cada unidade de processamento obtém do processador de controle a mesma instrução e carrega um elemento de dados separado por meio de seu acesso a dados privados da instrução que é executada. Desta forma, a instrução é executada de maneira síncrona em paralelo por todas as unidades de

processamento e para diferentes elementos de dados. GPU's encontradas em muitos computadores pessoais são exemplo de componentes de *hardware* construídos através deste modelo;

d) ***Multiple-Instruction, Multiple-Data* (MIMD):** Existem múltiplos elementos de processamento, cada qual com sua instrução e dados com acesso independente para memória de programa e de dados. A memória de dados pode ser compartilhada ou distribuída. A cada etapa, as unidades de processamento carregam as instruções armazenando os resultados em seus respectivos locais de memória. As unidades de processamento funcionam de modo assíncrono umas com as outras. Máquinas com processadores *multicore* e *clusters* são exemplos deste modelo (RAUBER; RUNGER, 2010).

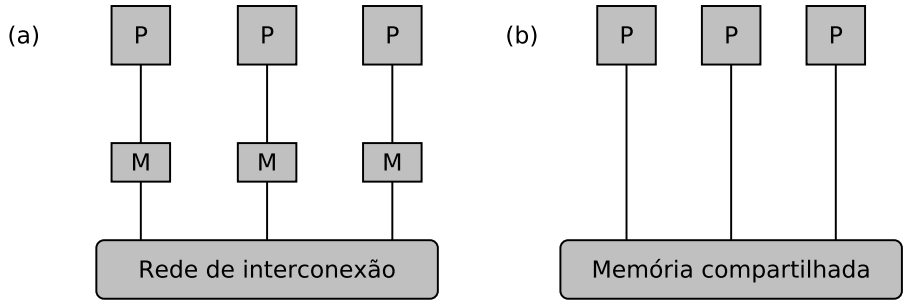
A maioria dos computadores de propósito geral são baseadas no modelo MIMD e podem ser classificados em duas categorias: memória distribuída e memória compartilhada. Máquinas de memória distribuída, descritas na Figura 3(a), são caracterizadas por possuírem uma memória (M) privada para cada unidade de processamento (P). A comunicação entre as unidades de processamento é realizada por meio do envio de mensagens de uma para a outra, através de uma rede de interligação e operações de comunicação explícitas. Esta arquitetura é encontrada em redes ou ambientes distribuídos tais como *clusters* ou grade computacional (DIAZ; MUNOZ-CARO; NINO, 2012). Por outro lado, máquinas de memória compartilhada, representadas na Figura 3(b), possuem uma memória global que armazena os dados da aplicação podendo ser acessada por todas as unidades de processamento. A comunicação entre as *threads* é realizada por meio do compartilhamento de variáveis, quando uma *thread* escreve e outra realiza a leitura do mesmo local (RAUBER; RUNGER, 2010). Máquinas de uso pessoal, *desktops* com processadores *multicore* enquadram-se nesta categoria, uma vez que os núcleos destes processadores compartilham a mesma memória principal (DIAZ; MUNOZ-CARO; NINO, 2012). Existem ainda máquinas híbridas, que utilizam aspectos tanto da arquitetura de memória distribuída quanto da memória compartilhada.

Um processador pode possuir várias unidades de processamento, que são conhecidas como núcleos. Diante disso, podem ser descritas duas abordagens de computadores de acordo com o número de núcleos incorporados (DIAZ; MUNOZ-CARO; NINO, 2012):

a) ***Multicore***: Máquinas cujo processador integra de 2 a 10 núcleos de processamento, mantendo a mesma velocidade de execução de programas sequenciais. Este tipo de processador é encontrado na maiora dos computadores pessoais atualmente;

b) ***Manycore***: Integram uma quantidade maior de núcleos de processamento, da

Figura 3 – Arquiteturas de computadores paralelos - Modelo MIMD



Fonte: dados da pesquisa

ordem de centenas. Máquinas *manycore* são exemplificadas pelas Unidades de Processamento Gráfico (*Graphical Processing Units* - GPU) e Xeon Phi.

Em virtude da diversidade de arquiteturas de computadores existentes, diversos modelos de programação paralela foram propostos. Segundo (DIAZ; MUNOZ-CARO; NINO, 2012), a programação paralela convencional envolve o modelo de memória compartilhada, com a API OpenMP em arquiteturas de memória compartilhada, ou um modelo de troca de mensagens usando a API MPI, em sistemas de memória distribuída. Por outro lado, a disponibilidade de computação de propósito geral em GPUs e em sistemas de múltiplos núcleos tem levado ao modelo de Programação Paralela Heterogêneo, buscando aproveitar as capacidades de CPUs *multicore* e de GPUs *manycore*. Assim, para arquiteturas híbridas, diferentes modelos de programação paralela podem ser combinados, dando origem ao que é chamado de Programação Paralela Híbrida (DIAZ; MUNOZ-CARO; NINO, 2012).

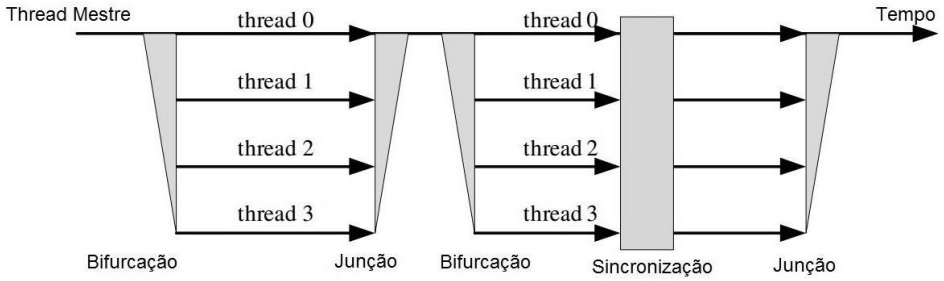
A referida API OpenMP é abordada na seção a seguir.

2.2.2 OpenMP

OpenMP é um padrão portável de programação para sistemas de memória compartilhada. A API OpenMP pode ser utilizada juntamente com as linguagens de programação C, C++ e Fortran e os recursos disponibilizados incluem diretivas de compilação, rotinas de biblioteca e variáveis de ambiente (RAUBER; RUNGER, 2010). Este modelo de programação é baseado na cooperação entre as *threads* executando instruções simultaneamente em múltiplos núcleos, onde as *threads* são criadas e destruídas de acordo com o padrão *fork-join* (RAUBER; RUNGER, 2010). A Figura 4 apresenta um esquema de execução que observa esse padrão. O processo inicia-se com uma única *thread master* que, ao alcançar uma região paralela, divide-se em novas *threads* para realizar o processamento paralelo das instruções (YOU; LEE; SUNG, 2009).

Os protótipos das funções e as definições paralelas OpenMP são providas através da

Figura 4 – Esquema de execução paralela no padrão *fork-join*.



Fonte: adaptado de (YOU; LEE; SUNG, 2009)

inclusão do arquivo de cabeçalho *omp.h* em C e C++, com o paralelismo sendo controlado por meio da inserção de diretivas de compilação (#pragma omp) (CHANDRA et al., 2001). A diretiva mais importante é a que define a região paralela do programa, com a seguinte sintaxe:

```
#pragma omp parallel [clausula[[,] clausula]... ] nova linha
{
        instrucoes...
}
```

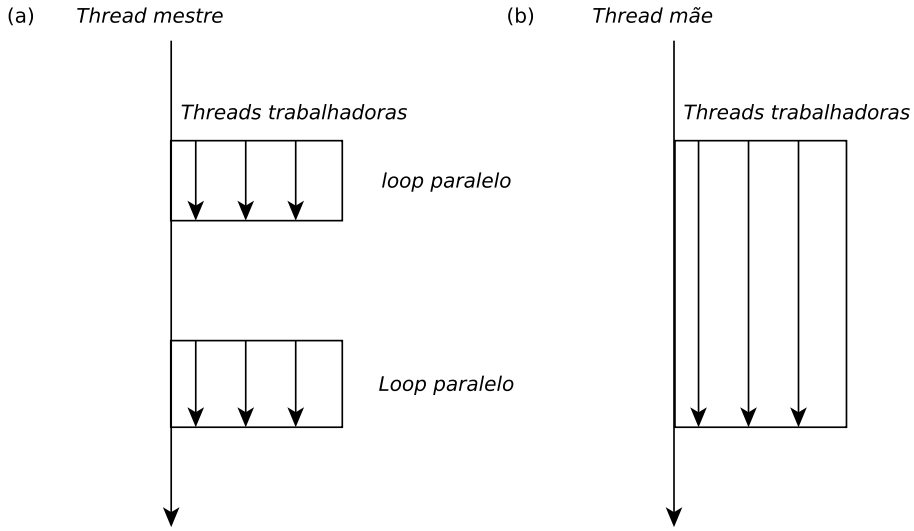
Listing 2.1 – Diretiva paralela

A diretiva *parallel* é utilizada para indicar que a região deve ser executada em paralelo. Desta forma, ao encontrar a diretiva, a *thread* mestre é desdobrada em novas *threads* trabalhadoras que passam a trabalhar de maneira independente dentro da região paralela. A cada uma das *threads* é atribuído um identificador único que vai de 0, identificador da *thread* mestre, ao número de *threads* menos um (RAJASEKARAN; REIF, 2008).

O OpenMP possui duas abordagens principais para atribuir trabalho à *threads*: *loop* e regiões paralelas. Na primeira abordagem, representada na Figura 5(a), cada *thread* assume uma gama de índices do *loop*, cujas tarefas são executadas em paralelo. Logo, o paralelismo concentra-se nos laços de repetição do programa, podendo o restante do código ser executado sequencialmente. Na abordagem de regiões paralelas, mostrada na Figura 5(b), todo o código pode ser executado paralelamente. A carga de trabalho dentro das regiões é distribuída entre as *threads* através de um identificador único atribuído a cada uma delas e a seleção é realizada por meio da estrutura de controle *if*, onde *my_id* representada o identificador atribuído à *thread*. Os exemplos 2.2 e 2.3 ilustram as abordagens por laço e região paralela, respectivamente.

No exemplo 2.2, a diretiva *for* informa ao compilador que o respectivo laço deve ser executado em paralelo, distribuindo as iterações entre as *threads*. Em 2.3, as instruções entre {} são executadas pela *thread* cujo identificador é igual *t*.

Figura 5 – OpenMP - Atribuição de trabalho às *threads*.



Fonte: dados da pesquisa

```
#pragma omp parallel for
for (int i = 0; i < N; i++){
        instrucoes...
}
```

Listing 2.2 – Laço de repetição

```
if(my_id == t){
        instrucoes...
}
```

Listing 2.3 – Regiões paralelas

O escalonamento das tarefas no OpenMP pode ser realizado manualmente através do código-fonte ou pelo próprio ambiente de desenvolvimento, em tempo de compilação ou de execução. Em um *loop* paralelo, na maioria das implementações, quando não se especifica nenhum tipo de escalonamento, as iterações são divididas igualmente entre as *threads* (CHANDRA et al., 2001). Essa forma de escalonamento é indicada para *loops* cujas iterações possuem carga de trabalho semelhante. Caso contrário, quando as iterações possuem carga de trabalho desbalanceada, poderá haver atraso devido à sincronização em algum momento do programa. Cargas de trabalho menores são processadas mais rapidamente, enquanto cargas maiores requerem maior tempo. Portanto, instruções rápidas, constituídas de cargas menores, precisam esperar o processamento de instruções mais lentas, com cargas de trabalho maiores. Como resultado, o tempo de execução do programa tende a aumentar (CHANDRA et al., 2001). Diante disso, existe a possibilidade de aplicar o escalonamento de acordo com as cargas de trabalho das instruções, visando reduzir

os atrasos causados pela sincronização. O escalonamento pode ser realizado de forma estática ou dinâmica:

a) **Escalonamento estático**: Nesse tipo de escalonamento, a distribuição do trabalho é realizada em função do número total de iterações do *loop* e do número de *threads*. Cada uma delas executa apenas as iterações que foram atribuídas no início do *loop*;

b) **Escalonamento dinâmico**: Diferentemente do escalonamento estático, a atribuição de trabalho pode variar durante a execução do *loop*. Nem todas as iterações são definidas logo no início. Dessa forma, cada *thread* pode solicitar novas tarefas em tempo de execução, à medida em que completa o trabalho atribuído.

Em ambos os esquemas, as iterações são atribuídas às *threads* em faixas contíguas, definidas como *chunks*. O **tamanho do *chunk*** é o número de iterações que cada *chunk* contém (CHANDRA et al., 2001). Por exemplo, um *loop* com 100 iterações executado por 4 *threads*, por padrão, terá *chunks* de tamanho 25 distribuídos para cada *thread*. Usando o escalonamento dinâmico, com tamanho de *chunk* 10, ao terminar de executar as iterações que foram atribuídas no início do *loop*, a *thread* pode solicitar outro *chunk* até que todos eles tenham sido processados.

As opções de escalonamento podem ser utilizadas atribuindo-se *chunks* às *threads* estática ou dinamicamente. A forma como os *chunks* são distribuídos pode ser caracterizada como **estático, dinâmico, guiado**:

a) **Estático (*static*)**: O *chunk* é atribuído estaticamente a cada *thread*. Se for especificado o tamanho, as iterações são divididas pelo tamanho do *chunk* e são distribuídos às *threads* de modo *round-robin*: a primeira *thread* recebe o primeiro chunk, a segunda recebe o segundo *chunk* e assim por diante, até que todos os *chunks* tenham sido distribuídos;

b) **Dinâmico (*dynamic*)**: As iterações são dividas pelo tamanho de *chunk* especificado e distribuídas às *threads* de maneira similar ao escalonamento estático. No entanto, a atribuição é realizada em tempo de execução de maneira dinâmica. Se não for especificado o tamanho do *chunk*, o tamanho padrão será 1;

c) **Guiado (*guided*)**: Semelhante ao escalonamento dinâmico, mas o tamanho do *chunk* diminui até o tamanho mínimo especificado. Por padrão, o tamanho inicial é aproximadamente: $\frac{N}{t}$, onde N é o número de iterações e t, o número de *threads*.

A cláusula para definir o escalonamento em OpenMP é dada na seguinte forma:

```
        schedule(tipo, chunk)
```

Listing 2.4 – Definição de escopo de variáveis

Onde o "tipo" pode assumir os valores *static*, *dynamic* e *guided*, sendo *chunk* um valor inteiro correspondente ao número de iterações dos blocos que serão atribuídos às *threads*.

A utilização do OpenMP é independente do sistema operacional e possui compiladores para todas as versões Unix e Windows. Portanto, a migração de um sistema para outro é uma questão de recompilação (CHANDRA et al., 2001). Além disso, um sistema escrito para executar em processadores com múltiplos núcleos pode ser executado em máquinas com núcleo único. As diretivas de programação paralela são ignoradas pelo compilador em processadores sequenciais e o programa é executado de forma sequencial. Além disso, existe a possibilidade de paralelismo incremental, quando o programador pode iniciar com o código sequencial e acrescentar aos poucos as diretivas que expressam a programação paralela.

Em relação ao compartilhamento de memória, a maior parte das variáveis no código OpenMP são visíveis a todas as *threads*, por padrão. Apenas índices de *loops* são privados. No entanto, são disponibilizadas cláusulas de gerenciamento do compartilhamento de memória. Tais diretivas possibilitam definir a visibilidade das variáveis em relação às *threads*, conforme especificado a seguir (CHANDRA et al., 2001):

a) ***Shared***: Representam locais de memória visíveis a todas as *threads* de modo que qualquer uma delas pode modificar e acessar. Esta é a condição padrão de todas as variáveis no OpenMP, com exceção dos índices dos *loops*;

b) ***Private***: Esta cláusula informa que as variáveis de uma região paralela são individuais para cada *thread*. Assim, cada *thread* terá uma cópia do local de memória;

c) ***Firstprivate***: Permite a inicialização de uma variável privada antes de entrar da região paralela;

d) ***Lastprivate***: Esta cláusula permite que o valor da variável ao final da região paralela seja retido, para ser utilizado posteriormente.

A sintaxe para definição do escopo de variáveis em relação à *threads* é dada da seguinte forma:

```
#pragma omp parallel <clausula>(lista de variaveis)
```

O compartilhamento de variáveis entre *threads* pode acarretar condições de disputa, ou seja, várias instruções podem acessar a mesma região de memória e retornar resultados imprevisíveis por causa da sequência em que são executadas. Desta forma, são necessárias medidas de controle para que o acesso ocorra de forma a resolver as dependências entre as instruções. Este assunto será abordado na seção a seguir.

2.2.3 Identificação de dependências

As instruções que compõem um problema a ser resolvido através de programação paralela podem ser independentes entre si ou manter um certo nível de dependência. Algumas instruções podem requerer resultados que são processados por outras instruções, o que implica na execução em uma ordem específica. Por exemplo, uma *thread* não pode realizar a leitura de um local de memória que ainda não teve a escrita finalizada. Assim, programas paralelos em sistemas de memória compartilhada necessitam de sincronização e coordenação entre as *threads* para que a execução possa ocorrer de maneira correta (RAUBER; RUNGER, 2010). Segundo (CHANDRA et al., 2001), sempre que uma instrução em um programa lê ou escreve um local de memória, e outra instrução escreve ou lê o mesmo local, é dito que existe uma dependência de dados entre as duas instruções naquele local de memória. Dependências podem ser detectadas através da análise de como as variáveis são utilizadas pelas instruções, conforme especificado em (CHANDRA et al., 2001):

a) A variável é somente leitura, ou seja, a possibilidade de que ela seja escrita dentro do *loop* é nula? Se sim, então não há dependência;

b) Caso a variável seja escrita dentro do *loop*, considere os locais de memória que compõem a variável e que são atribuídos dentro do mesmo como locais diferentes. Para cada local, é somente uma iteração que o acessa? Se sim, não existe dependência. Caso contrário, ela existe e deve ser resolvida para que o programa funcione corretamente na implementação paralela.

As variáveis globais e subrotinas chamadas dentro do *loop* também podem causar dependências. Os *loops* podem atribuir valores a variáveis globais externas através dessas chamadas, fazendo com que haja divergência em relação aos valores encontrados na execução sequencial do programa.

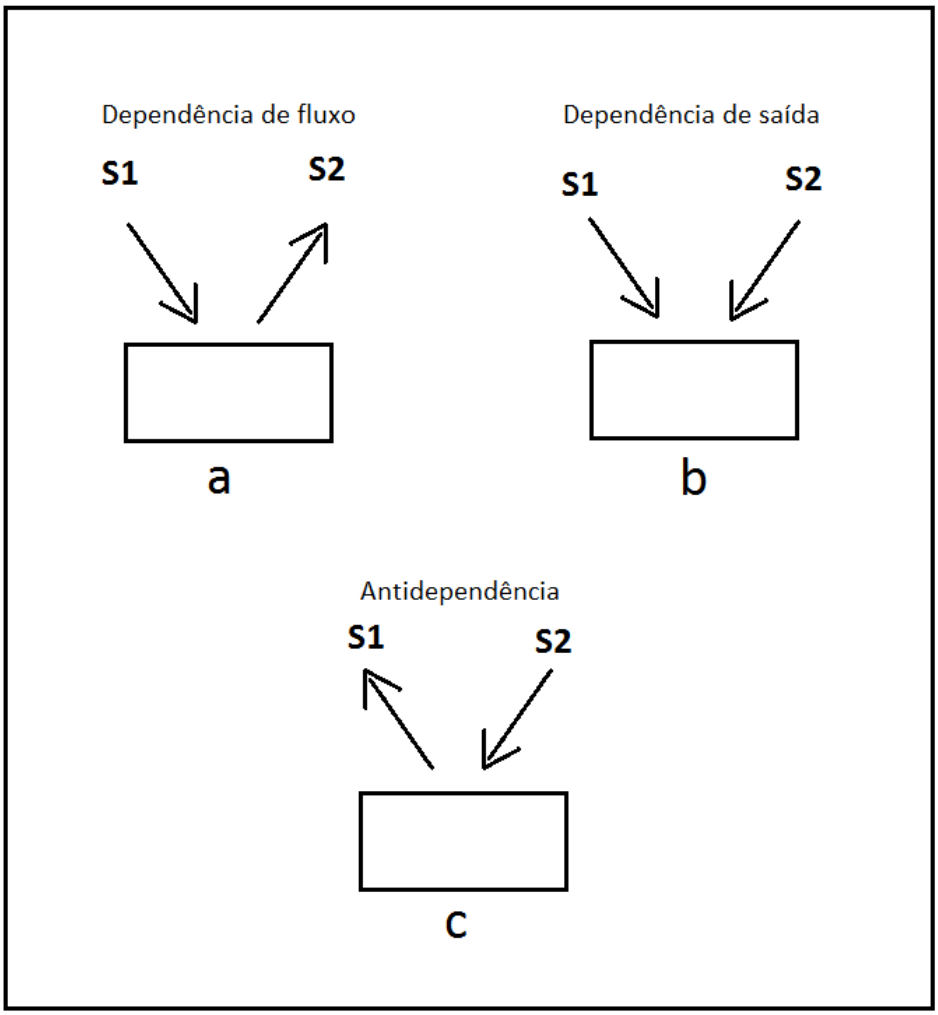
2.2.4 Classificação das dependências

Identificadas as dependências, é necessário classificá-las. Isso permite determinar se podem ser removidas ou como lidar com elas. (CHANDRA et al., 2001) ponderam

que as dependências podem ser classificadas com base no fato de serem ou não *loop-carried*, isto é, se as instruções envolvidas ocorrem ou não em iterações diferentes do *loop*. Dependências *loop-carried* causam concorrência de dados, porque dentro de uma única iteração de um *loop* paralelo, as instruções não são executadas em sequência e geram dependência entre diferentes iterações. Já aquelas *non-loop-carried* não necessitam de cuidados, pois são executadas sequencialmente dentro do *loop* a cada iteração, geralmente podem ser ignoradas.

Existe outro esquema de classificação de dependências crucial para a implementação paralela. É baseada no fluxo de dados entre as instruções dependentes, ou seja, trata-se da comunicação ou não por meio de locais de memória. A Figura 6 ilustra 3 tipos de dependência de fluxo de dados. Sejam S1 e S2 duas instruções diferentes, executadas por diferentes *threads*. Conforme descrito por (CHANDRA et al., 2001), podem ser identificadas entre elas Dependência de fluxo, Antidependência e a Dependência de Saída, que serão abordadas a seguir.

Figura 6 – Tipos de dependências segundo o fluxo de dados.



Fonte: dados da pesquisa

a) **Dependência de fluxo**: É o tipo de dependência mais importante. Está presente quando S1 escreve um local de memória e o valor lido por S2 em uma execução sequencial é o mesmo escrito por S1 (Figura 6a). O resultado de S1 "flui" para S2, por isso é denominada dependência de fluxo. Na maioria dos casos, este tipo de dependência não pode ser resolvido para que a execução possa ocorrer em paralelo, o que implica que obrigatoriamente, S1 deve executar antes de S2;

b) **Antidependência**: S1 lê um local de memória e S2 escreve no mesmo local. Ao contrário da dependência de fluxo, este tipo de dependência não representa comunicação de dados entre S1 e S2 (Figura 6b). Neste caso, o local de memória é compartilhado pelas instruções apenas a título de reuso. Os valores serão utilizados em momentos e pontos diferentes do programa;

c) **Dependência de saída** Neste tipo de dependência, ambas instruções S1 e S2 escrevem no mesmo local de memória (Figura 6c), o que faz com que uma variável possa apresentar valores diferentes em diferentes execuções do programa. Diante disso, as saídas do programa podem se mostrar corretas em alguns momentos, o que dificulta sua detecção.

2.2.5 Remoção de dependências

Identificados os tipos de dependências entre as instruções, procede-se com a remoção. Algumas delas podem ser removidas mudando-se o escopo das variáveis envolvidas, ou fazendo pequenas alterações no código. Cada uma delas tem um procedimento específico de resolução.

2.2.5.1 Remoção de Antidependências e Dependências de Saída

Conforme abordado na seção 2.2.4, as antidependências são caracterizadas por uma instrução (S1) efetuar a leitura de um local que é escrito por outra instrução (S2). Dependências desse tipo podem ser resolvidas dando a cada *thread* ou iteração uma cópia separada do local de memória. Se o local é inicializado a cada iteração antes que S1 leia, a dependência pode ser removida tornando-se privado este local de memória. Caso a variável seja inicializada fora do *loop*, uma cópia pode ser realizada e a leitura pode ser feita por meio da cópia, ao invés de utilizar a original, como demonstrado nos exemplos 2.5 e 2.6. O vetor a foi copiado em uma iteração à frente e armazenado no vetor $a2$. Assim, evita-se que dados de diferentes iterações sejam acessados em uma mesma iteração dentro do *loop* paralelo.

```
        for(int i = 1; i < N; i++) {
```

```
        x = (b[i] + c[i])/2;
        a[i] = a[i + 1] + x;
}
```

Listing 2.5 – Loop sequencial - antidependência no vetor a

```
#pragma omp parallel for shared(a, a2)
for(int i = 1; i < N; i++){
        a2[i] = a[i + 1];
}

#pragma omp paralell for shared(a, a2)
for(1 = 1; i < N; i++){
        x = (b[i] + c[i])/2;
        a[i] = a2[i] + x;
}
```

Listing 2.6 – Versão paralela com dependência removida

As variáveis globais, após o *loop* paralelo, devem ter os mesmos valores que teriam se a execução fosse realizada sequencialmente. No caso de variáveis com escopo compartilhado, essa condição é satisfeita quando o *loop* não possui dependências *loop carried*. Por outro lado, variáveis globais com escopo privado necessitam de finalização para garantir que terão os valores corretos quando o *loop* for finalizado. No exemplo 2.7, pode-se aplicar finalização em x alterando o escopo da variável para *lastprivate*. Isso faz com que seu escopo permaneça privado dentro do *loop*, mas os valores sejam atribuídos a ela e voltem para uma cópia de escopo compartilhado na última iteração. Já no caso de d[1], a mudança de escopo não funciona, pois na última iteração o valor de $d[2]$ será sobrescrito. Uma solução é a introdução de uma variável *lastprivate* e a atribuição a ela, do valor de $d[1]$, conforme apresentado em 2.8.

```
for(int i = 1; i <= N; i++){
        x = (b[i] + c[i])/2;
        a[i] = a[i] + x;
        d[1] = 2 * x; //loop carried em d
}
y = x + d[1] + d[2];
```

Listing 2.7 – Versão sequencial com dependência em $d\,[1[$

```
#pragma omp parallel for shared(a) lastprivate(x, d1)
for(int i = 1; i <= N; i++){
        x = (b[i] + c[i])/2;
```

```
                a[i] = a[i] +x;
                d1 = 2 * x;
        }
        d[1] = d1;
        y = x + d[1] + d[2];
```

Listing 2.8 – Versão paralela com dependência resolvida

2.2.5.2 Remoção de dependências de fluxo

Embora dependências de fluxo não permitam que instruções executem em paralelo, existem casos especiais que podem remover este tipo de dependência. O exemplo 2.9 mostra uma dependência de fluxo que pode ser resolvida através da aplicação de variáveis de redução.

```
int x = 0;
for(int i = 1; i <= N; i++){
        x = x + a[i];
}
```

Listing 2.9 – Versão sequencial: dependência de fluxo em x

Neste caso, a instrução que atualiza a variável x também realiza a leitura, causando dependência de fluxo. A aplicação de redução nesta variável resolve o problema, como pode ser visto em 2.10.

```
x = 0;
#pragma omp parallel for reduction(+: x)
for(int i = 1; i <= N; i++){
        x = x + a[i];
}
```

Listing 2.10 – Versão paralela - depedência resolvida

Quando uma variável é atualizada dentro do *loop* da mesma forma como uma redução, mas usa o valor atualizado em outro local de dentro do *loop*, a dependência não pode ser resolvida simplesmente com a aplicação de redução. O Exemplo 2.11 ilustra este tipo de dependência:

```
//Versao sequencial
float idx = N/2 + 1;
int i_sum = 1;
float pow2 = 2;

for(int i = 1; i <= N/2; i++){
```

```
        a[i] = a[i] + a[idx];
        b[i] = i_sum;
        c[i] = pow2;
        idx = idx + 1;
        i_sum = i_sum + 1;
        pow2 = pow2 * 2;
}
```

Listing 2.11 – Versão sequencial com depdendências em idx, i_sum e $pow2$

Percebe-se que idx, i_sum e $pow2$ são atualizados dentro do *loop*. Neste caso, os valores destas variáveis diferem ao longo da execução paralela se comparados à execução sequencial. Para resolver a dependência, utiliza-se de uma classe especial de redução chamada "indução", em que o valor da variável de redução durante cada iteração é uma função do índice da variável no *loop*. Esta técnica é denominada "eliminação de variável de redução", pois há substituição de variáveis que são atualizadas por meio de multiplicação por constante, incremento por constante ou função de um *loop* por uma expressão que contém o índice do *loop*. No exemplo 2.12, as variáveis idx, i_sum e $pow2$ do código sequencial apresentado em 2.11 foram substituídas por suas respectivas funções.

```
#pragma omp parallel for shared(a, b, c)
for(int i = 1; i <= N/2; i++){
        a[i] = a[i] + a[i + N/2];
        b[i] = i * (i + 1)/2;
        c[i] = 2 ** i;
}
```

Listing 2.12 – Versão paralela com dependências resolvidas.

Outra técnica utilizada na remoção de dependências deste tipo é chamada *loop skewing*, cuja ideia é converter um *loop* com dependência de fluxo do tipo *loop-carried* para um do tipo *non-loop-carried*. Veja o exemplo 2.13.

```
for(int i = 2; i <= N; i++){
        b[i] = b[i] + a[i - 1];
        a[i] = a[i] + c[i];
}
```

Listing 2.13 – Versão sequencial com dependência no vetor a

Neste exemplo existe uma dependência *loop-carried* da atribuição de $a[i]$ na iteração i para a leitura de $a[i-1]$ na iteração $i+1$. Como os elementos de a não são dependentes entre si, podem ser processados em paralelo. Logo, foi realizado o "deslocamento" da leitura de $a[i]$ na iteração $i+1$ para a iteração i. Assim, a dependência se tornou do tipo *non-loop-carried*. Finalmente, os índices e os limites do *loop* foram

ajustados devidamente. Percebe-se, portanto, que com o deslocamento de b, não será mais necessária a leitura de $a[i-1]$ na iteração $i+1$. As alterações realizadas no código sequencial são apresentadas no Exemplo 2.14.

```
b[2] = b[2] + a[1];
#pragma omp parallel for shared(a, b, c)
for (int i = 2; i <= N - 1; i ++){
        a[i] = a[i] + c[i];
        b[i + 1] = b[i + 1] + a[i];
}
```

Listing 2.14 – Versão paralela com dependências resolvidas.

3 TRABALHOS RELACIONADOS

A implementação paralela da extração de atributos é investigada com fins de utilização em diversas áreas, inclusive Reconhecimento Automático de Fala. Como será visto adiante, os esforços em melhorar o desempenho focam muito na utilização de diferentes tipos de plataformas paralelas de *hardware*, tais como processadores *multicore*, caracterizados por possuírem várias unidades de processamento em um mesmo *chip*, FPGA (*Field Programmable Gate Array*) e GPU (*Graphics Processing Unit*), dentre outros.

Um trabalho de destaque na implementação paralela da extração de atributos foi realizado por (MAKA; DZIURZANSKI, 2013). Construiu-se uma biblioteca para extração dos atributos de áudio utilizados em um sistema de busca e indexação de som. Assim como no presente trabalho, foi utilizada a API de programação paralela OPenMP em arquitetura *multicore* na implementação. Os testes na biblioteca de (MAKA; DZIURZANSKI, 2013) foram realizados em uma máquina com processador Intel(R) Core(TM)2. Ao passo que este trabalho realizou testes para três tipos de escalonamento (estático, dinâmico e guiado), (MAKA; DZIURZANSKI, 2013) utilizaram o escalonamento estático, o que proporcionou menor custo de sincronização devido ao tipo de carga processada, que se apresentou balanceada. Houve ganho de desempenho de aproximadamente 15% no escalonamento estático em relação ao escalonamento dinâmico. Além disso, diferentes tamanhos de *frames* foram testados (entre 128 e 1024 amostras) em processadores de 1 a 4 núcleos. Percebeu-se que o ganho em relação à implementação sequencial cresce conforme aumenta-se o tamanho do *frame*.

O mesmo trabalho ainda avaliou o comportamento do sistema quanto à utilização de 32 núcleos. Para isso, o sistema foi executado em uma máquina Linux 64-bit com processador Intel(R) Xeon(R) CPU E7310 1.60GHz e 32 núcleos com 2048 KB de cache. O tamanho do *frame* foi fixado em 1024 amostras. Os resultados mostram que há ganho de desempenho de 60% em relação à implementação sequencial com a utilização de até 15 núcleos, a partir disso, o ganho decresce. Verificou-se comportamento semelhante em tamanhos de *frames* menores, com a diferença de que o número de núcleos com máximo desempenho varia de acordo com o tamanho do *frame*. Segundo os autores, isso ocorre devido à característica dos *loops*, que realizam operações aritméticas simples. Essas operações não oferecem carga de trabalho suficiente para compensar os custos com comunicação e a sobrecarga de sincronização entre os núcleos. Para um número pequeno de núcleos, este fato não é observado.

Três técnicas de extração de atributos foram paralelizadas em (MAJID; MIRZAEI; JAMALI, 2012): *Fast Fourier Transform* (FFT), *Mel-Frequency Ceptral Coefficient* (MFCC) e *Discrete Wavelet Transform* (DWT). A pesquisa utiliza Rede Neural Evolucionária (*Parallel Evolutionary Neural Network* - ENN) paralela como classificador.

A ENN combina a Rede Neural *Feed Forward* com Algoritmo Genético (*Genetic Algorithm* - GA), sendo que esse último tem mostrado grande efetividade para otimização de funções e pode eficientemente buscar grandes e complexos espaços de busca para encontrar a solução ótima mais próxima. O objetivo dessa implementação é extrair recursos de forma mais eficiente com menor tempo de execução. A aplicação foi desenvolvida para arquitetura com múltiplos núcleos de processamento, em C# e utilizou as APIs para implementação *multithread* disponíveis na própria linguagem. O presente estudo utilizou a extração de atributos baseada em coeficientes MFCC por proporcionar extração da informação do áudio de maneira compacta, em comparação às demais técnicas descritas acima.

No estudo realizado em (BAHOURA; EZZAIDI, 2013), foi implementado em FPGA o método de extração de atributos baseado em MFCC para a utilização em um sistema de análise e classificação de sons da respiração em tempo real. A proposta utilizou o ambiente Simulink do Matlab e o Sistema Gerador Xilinx (XSG) em duas abordagens: ponto-fixo e ponto-flutuante. Os vetores de atributos obtidos com a implementação de ponto fixo se comparam aos obtidos com a implementação do Matlab usando ponto flutuante, sendo que ambos utilizaram sons de respiração asmática e de respiração normal.

Ainda em 2013, (KOU et al., 2013) utilizaram GPU para a implementação de um extrator de atributos também baseado em MFCC para um ASR. Foi demonstrado que esse tipo de hardware é adequado à geração de vetores MFCC e houve uma redução substancial no tempo de processamento, comparando-se com a implementação sequencial em uma CPU. A concorrência de dados dentro de um *frame* individual foi mapeada para um bloco de *threads* e a concorrência entre *frames* diferentes foi mapeada para blocos de *threads* diferentes, o que permitiu interações eficientes entre as *threads*. Para referência de resultados, os autores utilizaram uma adaptação do software de reconhecimento de voz Julius [*]. O sistema paralelo mostrou-se 90x mais rápido, possibilitando que a extração de recursos fosse realizada com fator 0,01% de tempo real (RTF – *Real Time Factor*). Diante disso, 2 horas de áudio puderam ser extraídas em segundos.

Este trabalho implementa a extração de atributos MFCC de um ASR de palavras isoladas, utilizando a arquitetura de computadores de múltiplos núcleos e a API de programação paralela OpenMP.

[*] http://julius.sourceforge.jp/en_index.ph

4 IMPLEMENTAÇÃO PARALELA DA EXTRAÇÃO DE ATRIBUTOS

O sistema desenvolvido neste trabalho é baseado na implementação sequencial de um ASR de palavras isoladas baseado em HMM, desenvolvido pelo aluno Willian Antônio dos Santos, do curso de Ciências da Computação da Pontifícia Universidade Católica de Minas Gerais. Na implementação da versão paralela do sistema, foi utilizada a linguagem C e a API de programação paralela OpenMP em ambiente Linux. As bibliotecas sequenciais foram modificadas para que o ASR pudesse ser executado em plataforma *multicore*, realizando o processamento das etapas utilizando múltiplas *threads*. Portanto, os módulos relativos à extração de atributos do sistema sequencial foram analisados para identificação das dependências e adaptados para execução paralela.

O desenvolvimento da aplicação foi organizado em 2 etapas:

a) **Análise do código sequencial**: essa etapa tem por objetivo identificar as dependências do código sequencial para que a adaptação dos módulos para execução paralela possa ser realizada;

b) **Implementação paralela**: compreende a realização dos procedimentos para resolver as dependências identificadas na etapa anterior.

A seção a seguir inclui os detalhes das etapas abordadas.

4.1 Análise do código sequencial

A análise do código sequencial propõe o levantamento dos pontos de dependência de dados de forma a possibilitar sua resolução durante a implementação da versão paralela do sistema. É importante ressaltar que o sistema abordado nesse trabalho utiliza bibliotecas do HTK (DEPARTMENT, 1993) para gerenciamento de memória, manipulação de áudio e processamento de sinais, cujas funções são descritas a seguir:

a) **HMem**: Contém as estruturas e funções de gerenciamento de memória em baixo nível. Essa biblioteca é responsável pela alocação dos recursos utilizados na representação das amostras de áudio e dos coeficientes MFCC;

b) **HWave**: Representa as estruturas e funções de manipulação de áudio. As operações de leitura e armazenamento das amostras, conversão de formatos e carregamento do áudio em quadros sobrepostos são realizadas por essa biblioteca;

c) **HSigP**: Conjunto de estruturas e funções para processamento de sinais. Através desse módulo, é possível realizar as principais etapas do *Front-end*: aplicação do

filtro de pré-ênfase, da janela de Hamming, do banco de filtros e cálculo da energia dos mesmos, aplicação da FFT (*Fast Fourier Transform*) e da DCT (*Discrete Cosine Transform*), conforme descrito na seção 2.

As bibliotecas descritas compõem os módulos do sistemas sequencial e foram adaptadas para execução paralela. O Algoritmo 1 apresenta a descrição do funcionamento do sistema sequencial como um todo. As etapas modificadas para funcionamento de maneira paralela são mostradas em *itálico*.

Conforme apresentado, o áudio adquirido é dividido em N blocos de tamanho $\tilde{f}rameSize$ que são processados sequencialmente. Esses blocos são armazenados em um *buffer* que é processado nas etapas de aplicação do filtro pré-ênfase, janelamento, aplicação da FFT, banco de filtros, cálculo do log de energia e aplicação do Transformada Discreta do Cosseno (DCT). Em seguida, o resultado do processamento deste *buffer* é armazenado em um vetor denominado MFCC, que poderá ter anexado em sua última posição o coeficiente 0. O vetor MFCC resultante é destinado ao cálculo das derivadas, cujo resultado constitui entrada para a etapa de reconhecimento de padrões, responsável por identificar a palavra.

```
1: Divide audio em N blocos;
2: for i = 1 to N do;
3:     buffer=bloco[i];
4:     Aplica pré-ênfase;
5:     Aplica janela de Hamming;
6:     Aplica FFT;
7:     Aplica Banco de filtros;
8:     Calcula log energia;
9:     Aplica DCT;
10:    Armazena coeficientes MFCC[ ];
11:    if tem Coeficiente 0 then
12:        Calcula coef0;
13:        MFCC[tamanho] = coef0;
14:    end if
15:    Calcula derivada primeira
16:    Calcula derivada segunda
17:    Viterbi(MFCC)
18: end for
```

Algoritmo 1 – Extração de atributos MFCC

Fonte: Dados da pesquisa.

O processamento dos blocos de áudio é realizado através de um *loop* que percorre as janelas de áudio por meio de um *buffer*. Uma vez que o *buffer* pode ser acessado em diversas iterações e dentro dele existem funções que alteram os dados processados, podem ocorrer dependências entre elas. Conforme apontado por (CHANDRA et al., 2001), a característica chave de um *loop* para que execute corretamente em paralelo, é ausência

de dependência de dados. Logo, foram necessárias adaptações para que a extração de atributos pudesse ser executada de maneira paralela e garantir assim, a fidelidade dos resultados em conformidade com a implementação sequencial do sistema. Foram encontradas dependências nas etapas de aquisição do áudio, aplicação da janela de Hamming e preenchimento das informações dos filtros triangulares. As etapas de Aplicação do filtro Pré-ênfase, da FFT, da DCT e do cálculo do Log de energia não apresentaram dependência de dados, uma vez que os cálculos são realizados em nível de amostra dentro do bloco, não dependendo de estruturas externas para a realização das operações. As alterações propostas na implementação paralela das etapas que apresentaram dependências são descritas nas seções a seguir.

4.2 Implementação do processo de aquisição do áudio

O processo de aquisição do áudio no ASR é descrito na Figura 7. O carregamento das amostras para a memória na versão sequencial é realizado através do preenchimento de N *frames* de tamanho *frameSize*. Para o preenchimento dos blocos na versão sequencial, utiliza-se um ponteiro p que percorre o áudio recebido a cada t_frame amostras, onde t_frame corresponde à taxa do *frame*. Esse ponteiro é atualizado a cada iteração, ou seja, a cada novo bloco carregado na memória:

```
        p += t_frame
```

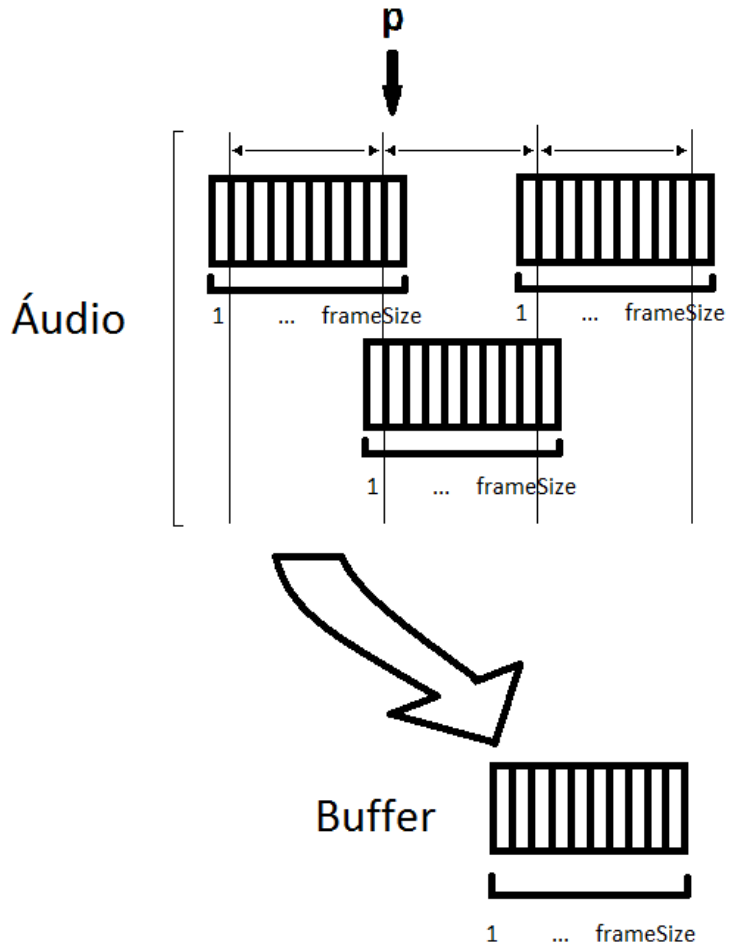
O tamanho do *buffer* de áudio copiado na realização da extração dos atributos está em torno de 250 amostras e o *overhead* causado pela cópia se torna insignificante em relação ao tempo de execução do programa, portanto, foi desconsiderado.

No entanto, na implementação paralela, o compartilhamento do ponteiro p pelas *threads* acarreta dependência de saída, quando várias instruções escrevem no mesmo local (CHANDRA et al., 2001). Dessa forma, uma *thread* poderia alterar o valor de p utilizado por outra *thread*, que se encontra trabalhando em local diferente do áudio, fazendo com que a sequência de amostras fosse processada em ordem divergente da versão sequencial. Para resolver o problema, a solução proposta neste trabalho é a realização de uma cópia da estrutura que guarda o áudio para cada *thread*, de modo que cada uma delas tenha sua cópia do ponteiro e possa atualizar seu valor efetuando o carregamento do áudio sem que haja interferências de outras *threads*. Além disso, o valor de p passou a ser atualizado da seguinte maneira:

```
p = ((frmAtual - 1) * t_frame);
```

Com isso, o ponteiro guarda também a localização do bloco no vetor, uma vez que as *threads* realizam o processamento em diferentes regiões do áudio simultaneamente.

Figura 7 – Processo de aquisição do áudio.



As amostras são carregadas na memória através de um *buffer* de tamanho *frameSize*. O ponteiro p identifica a amostra atual.

Fonte: dados da pesquisa

4.2.1 Implementação da aplicação da Janela de Hamming e preenchimento do banco de filtros

A janela de Hamming tem por objetivo suavizar o sinal janelado, evitando o fenômeno de Gibbs. Essa estrutura é implementada na biblioteca HSigP. Nessa implementação, todos os blocos de voz são multiplicados pela mesma janela no decorrer do programa. Desta forma, para sua criação, foram utilizadas estruturas que garantem que seus dados estejam presentes na memória durante todo o escopo, ou seja, foram utilizadas funções e variáveis estáticas. Embora a janela deva estar presente na memória durante a execução do programa, seu tamanho varia de acordo com o tamanho do bloco que está sendo multiplicado. Portanto, esse tamanho deve ser ajustado, o que implica na realocação do espaço de memória ocupado pela janela. Na versão sequencial isso não acarreta problemas, no entanto, na execução paralela as várias *threads* tentam alocar um espaço já alocado, ocasionando erro de execução. Diante disso, para resolver o problema, foi realizada a replicação

da janela para cada *thread*, conforme apresentado nos exemplos 4.1 e 4.2. Assim, cada *thread* possui sua cópia da janela, eliminando o problema de compartilhamento de regiões de memória.

```
//VERSAO SEQUENCIAL
vector janela;
janela[i] = createVector(frameSize);
```

Listing 4.1 – Criação da Janela de Hamming - versão sequencial

```
//VERSAO PARALELA
int n_threads;
vector * janela;
/*Cria o vetor com a copia da janela para cada thread*/
janela = (vector)malloc(sizeof(vector) * n_threads);
/*cria um vetor janela para cada uma das threads*/
for(int i = 0; i < n_threads; i++)
        janela[i] = createVector(frameSize);
```

Listing 4.2 – Criação da Janela de Hamming - versão paralela

Em relação às informações do banco de filtros, também foi constatada dependência de saída no preenchimento das informações por cada *thread*. Assim como na solução apresentada para a aplicação da janela de Hamming, foram criadas cópias das estruturas que guardam as informações do banco de filtros, com o objetivo de permitir a cada *thread* a gravação das informações sem que haja interferência das demais. Esse processo é ilustrado no código apresentado em 4.3.

```
//VERSAO PARALELA
infoFiltros filtro;
int n_threads;
filtro = (infoFiltros*)malloc(n_threads * sizeof(infoFiltros));
        for(th = 0; th < n_threads; th++){
                filtro[th] = inicializaBanco(frameSize, (*
                   peridoAmostras), numCanais, ...);
        }
```

Listing 4.3 – Criação da estrutura que armazena as informações do banco de filtros

4.3 Funcionamento do sistema paralelo

Na versão paralela do sistema, assim como na versão sequencial, o áudio adquirido é dividido em N blocos que são armazenados em *buffers* e processados pelas *threads* a cada

iteração do *for*. O número total de blocos é dividido entre elas realizando o processamento em paralelo. O Algoritmo 2 descreve a implementação paralela do sistema utilizando as diretivas OpenMP.

```
   int frameSize;
   Divide audio em N blocos;
1: #pragma omp parallel private(lista de variaveis) firstprivate(lista de variaveis)
2: #pragma omp single {
3: cria janela de Hamming(frameSize, n_threads)
4: aloca estruturas do banco de filtros(infoBanco, n_threads)
5: }
       #pragma omp for schedule(tipo, chunk)
6: for i = 1 to N do;
7:     buffer=bloco[i];
8:     Aplica pré-ênfase;
9:     Aplica janela de Hamming;
10:    Aplica FFT;
11:    Aplica Banco de filtros;
12:    Calcula log energia;
13:    Aplica DCT;
14:    Armazena coeficientes MFCC[ ];
15:    if tem Coeficiente 0 then
16:        Calcula coef0;
17:        MFCC[tamanho] = coef0;
18:        Copia buffer(bloco)
19:    end if
20: end for
21: for i = 1 to N do
22:    Calcula derivada primeira(bloco[i])
23:    Calcula derivada segunda(bloco[i])
24:    Viterbi(bloco[i])
25: end for
```

Algoritmo 2 – Implementação paralela do sistema

Fonte: Dados da pesquisa.

Como é possível verificar, a primeira etapa consiste na divisão do áudio em N blocos de tamanho $frameSize$. O início da região paralela é definido pelas diretivas *#pragma omp paralell*, onde também são definidos os escopos das variáveis utilizadas nessa região. Em seguida, é criada a janela de Hamming e alocadas as estruturas que guardam as informações do banco de filtros, estruturas que são replicadas para cada *thread*. A etapa de alocação dessas estruturas deve ser realizada apenas uma vez. Para isso, utilizou-se a diretiva *single*, que especifica que aquele ponto deve ser executado por apenas uma das *threads*.

O processamento dos blocos é realizado através do for que vai de 1 a N, sendo N o número total de blocos do sinal de voz. As diretivas *#pragma omp for schedule* especificam que a execução desse for deve ocorrer em paralelo, utilizando o escalonamento

especificado em *tipo* e tamanho de *chunk* definido. Os blocos são, portanto, processados pelas *threads*, que realizam as etapas descritas na seção 2.1 para cada bloco em paralelo. O resultado deste processamento é armazenado em vetores MFCC que são copiados para uma estrutura que processa as derivadas primeira e segunda, passando ao reconhecimento através do Algoritmo de Viterbi. Esta última etapa, ou seja, o cálculo das derivadas, por apresentar significativa dependência de dados, foi implementada sequencialmente.

Para manter a confiabilidade dos resultados retornados pela implementação paralela, as saídas do programa paralelo foram comparadas com as saídas do programa sequencial em cada uma das etapas, de modo a manter o mesmo padrão. Desta forma, a taxa de acerto no reconhecimento das palavras no sistema paralelo é a mesma obtida para o sistema sequencial.

5 METODOLOGIA

Este capítulo tem por objetivo apresentar o método experimental utilizado na realização deste trabalho, que compreende as seguintes etapas:

a) Definição das métricas de desempenho;

b) Configuração do ambiente de testes;

c) Realização dos testes experimentais.

Os modelos utilizados no reconhecimento das palavras foram treinados por meio do conjunto de ferramentas HTK, que foi desenvolvido originalmente no Departamento de Engenharia de Cambridge cujo código-fonte, na linguagem C, é disponibilizado através do endereço http://htk.eng.cam.ac.uk. Os detalhes da realização do estudo são descritos nas seções a seguir.

5.1 Métricas de desempenho

As métricas de desempenho utilizadas neste trabalho consistem na medição do tempo de execução do sistema, cálculo do *speedup* e utilização de contadores de *hardware*.

5.1.1 Tempo de execução e speedup

Uma das métricas de desempenho de aplicações paralelas é o tempo de execução paralelo (T_p). Essa métrica representa o espaço de tempo entre o início da primeira tarefa executada pelo primeiro processador e o final da última tarefa executada pelo último processador (RAJASEKARAN; REIF, 2008). Através do tempo de execução é possível verificar o ganho de velocidade de um programa paralelo em relação ao tempo de execução da versão sequencial do programa. Assim, é possível estabelecer uma relação entre eles. Essa métrica é definida como *speedup* (S), e é calculada através da razão entre o tempo de execução da versão sequencial (T_s) e o tempo de execução da versão paralela (T_p), sendo utilizada para representar o aumento de velocidade de execução de um programa em relação a outro, conforme descrito na Equação 12.

$$S = \frac{T_s}{T_p} \tag{12}$$

5.1.2 Desempenho de hardware

Os processadores utilizados neste trabalho possuem registros internos com capacidade para contar a ocorrência de eventos que fornecem informações detalhadas sobre

a interação entre a aplicação e o *hardware*. Tais registros são definidos como contadores de *hardware* e permitem obter registros internos da CPU. Para isso, foi utilizada a API PAPI (*Performance Application Programming Interface*) (AL, 2015). Esta API especifica um padrão de programação para acessar os contadores de desempenho do *hardware* disponíveis na maioria dos processadores modernos. Os eventos medidos são armazenados em um vetor denominado *EventSets* e são descritos a seguir:

a) PAPI_L3_TCA: Número de acessos à cache L3;

b) PAPI_L3_TCM: Falta de cache L3;

c) PAPI_TOT_INS: Número de Instruções completadas;

d) PAPI_TOT_IIS: Número de Instruções emitidas;

e) PAPI_BR_CN: Número de instruções de desvio condicionais previstas corretamente;

f) PAPI_BR_PRC: Total de instruções condicionais;

g) PAPI_TOT_CYC: Total de ciclos;

h) PAPI_RES_STL: Ciclos paralisados aguardando recurso.

De acordo com os eventos listados acima, é possível extrair as seguintes métricas:

$$tp = \frac{PAPI\_BR\_CN}{PAPI\_BR\_PRC} \tag{13}$$

$$cp = \frac{PAPI\_RES\_STL}{PAPI\_TOT\_CYC} \tag{14}$$

$$ce = \frac{PAPI\_L3\_TCM}{PAPI\_L3\_TCA} \tag{15}$$

$$ic = \frac{PAPI\_TOT\_INS}{PAPI\_TOT\_IIS} \tag{16}$$

Onde:

a) tp = taxa de previsões corretas;

b) cp = taxa de ciclos paralisados;

c) ce = taxa de perda de cache L3;

d) ic = taxa de instruções completadas.

Em relação à métrica *tp*, apresentada na Equação 13, é importante mencionar o preditor de desvio condicional. Um preditor de desvio condicional é um circuito que tenta prever o caminho em uma estrutura condicional, como por exemplo, if-else. Um salto condicional pode fazer com que a instrução imediatamente após a atual seja executada, ou pode ocasionar um salto para outro local do código. A previsão de desvio de condicional faz com que a instrução mais provável de ser executada em um salto condicional seja colocada na fila no *pipeline* antes que a decisão seja tomada, ou seja, antes que o salto seja executado. Desta forma, não há desperdício de tempo devido à espera da decisão, podendo a instrução ser executada mais rapidamente. Por outro lado, caso haja erro na previsão do condicional, o fluxo correto deve ser retomado, o que acarreta atraso na execução da instrução. Portanto, quanto maior a taxa de acerto, melhor o desempenho do programa.

A métrica *cp*, mostrada na Equação 14, diz respeito à taxa de ciclos de *clock* em que uma instrução fica ociosa aguardando a liberação de algum recurso. Quanto maior o valor, mais ocioso está o programa. Desta forma, para melhor aproveitamento do *hardware* é interessante a diminuição desta taxa.

A métrica *ce*, apresentada na Equação 15, informa a taxa de falta de *cache*. Quando a CPU precisa de um dado da memória, a busca é realizada primeiramente na memória *cache*. Se o dado não estiver presente, denomina-se falta de cache (*cache miss*) e a partir daí, a busca é feita na memória principal, quando é coletado e finalmente armazenado na memória *cache*. Pelo fato de a memória principal ser mais lenta, esta busca implica em certo atraso no tempo de execução do programa.

Por fim, a métrica *ic*, apresentada na Equação 16, informa a taxa de instruções do programa que foram completadas em relação ao total de instruções buscadas. Portanto, é interessante que essa taxa tenha valores reduzidos.

5.2 Configuração do ambiente de testes

Os arquivos de áudio utilizados no trabalho possuem taxa de amostragem de 8kHz e resolução de 16 bits. Esses arquivos foram divididos em 2 grupos: um grupo foi utilizado na etapa de treinamento e outro na etapa de reconhecimento. A configuração dos modelos utilizados é mostrada no Quadro 1.

Quadro 1: *Configuração dos modelos*

Vetor de atributos - Número de coeficientes	12
Cálculo de Coeficiente 0	sim
Derivada primeira	sim
Derivada segunda	sim

Fonte: Dados da pesquisa

Para avaliação do sistema foram aplicados os escalonamentos estático, dinâmico e guiado. Para cada tipo de escalonamento, variou-se o tamanho de *chunk*, correspondente ao número iterações do *loop* para processamento de blocos de áudio, em potências de 2 (2^n, sendo $0 \leq n \leq 5$), além de variar o número de *threads*. Este último parâmetro foi configurado de acordo com o *hardware* utilizado, conforme especificado no Quadro 2.

Quadro 2: *Hardware* e número de *threads* utilizados

Máquina	Processador	Núcleos físicos	Memória	*Threads*
1	Intel Xeon(R) CPU E5645 2.40GHz	6	32GB	6
2	Intel Core(TM) i5-4200U CPU 1.60GHz	2	4GB	4

Fonte: Dados da pesquisa

5.3 Descrição dos testes

O sistema foi testado em ambiente Linux, utilizando modelos previamente treinados por meio do conjunto de ferramentas HTK. A avaliação foi realizada por meio da medição do tempo de execução e do desempenho através do acesso aos contadores de *hardware* disponíveis nos processadores.

Para obtenção do tempo médio de execução da extração de atributos foram realizadas 10 execuções do sistema para cada configuração descrita na seção 5.2. Para os testes na máquina 1, o número de *threads* variou de 1 a 4. Na máquina 2, foram realizados testes com o número de *threads* variando de 1 a 6.

6 RESULTADOS EXPERIMENTAIS

O sistema desenvolvido foi avaliado quanto ao tempo de execução da extração de atributos, variando-se o tipo de escalonamento, número de *threads* e tamanho de *chunk*. Foram realizadas análises variando-se o tamanho de *chunk* de 1 a 32, no entanto, os resultados intermediários ($chunk = 2^n, 1 \leq n \leq 4$) foram omitidos pelo fato de apresentarem pequena variação (menos de 5 %), sendo reportados neste trabalho, somente os resultados para *chunk* de tamanhos 1 e 32. Este capítulo apresenta os resultados obtidos.

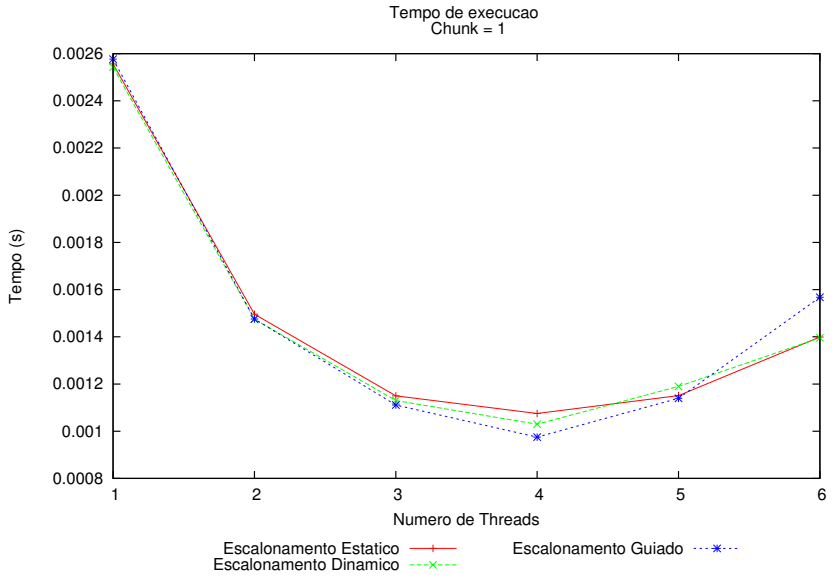
6.1 Resultados quanto ao tempo de execução e *speedup*

Os resultados alcançados são descritos nas seções a seguir. O valor mostrado nos gráficos 1, 2, 3 e 4 é resultado da média aritmética do tempo de execução da extração de atributos para reconhecimento de uma palavra. Para cada configuração, foram realizadas 10 execuções com número de *threads* variando de 1 a 6 e de 1 a 4 para as máquinas 1 e 2, respectivamente. O *Speedup* foi calculado com base no tempo de execução do sistema sequencial em cada *hardware*. Foram avaliados *chunks* de tamanhos 1 e 32.

6.1.1 Tempo de execução

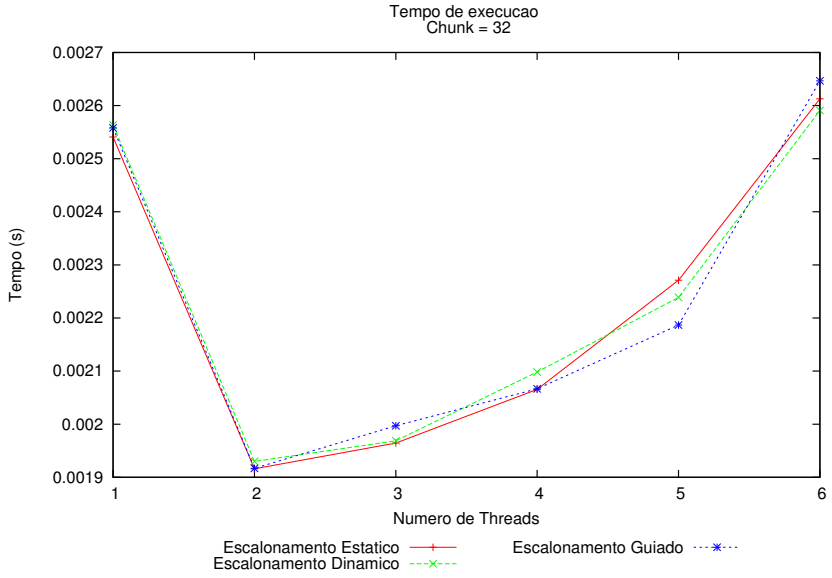
Os Gráficos 1 e 2 apresentam o tempo de execução do ASR em função no número de *threads* para a Máquina 1. Percebe-se que os menores tempos de execução para os escalonamentos testados foram alcançados com a execução do sistema utilizando-se entre 2 e 5 *threads*. Nestas regiões o *speedup* médio variou de 1,27 a 2,45 em relação à implementação sequencial. Para *chunk* de tamanho 1, demonstrado no Gráfico 1, foi alcançado menor tempo utilizando-se 4 *threads* e escalonamento dinâmico. Já para $chunk = 32$, ilustrado no Gráfico 2, a utilização de duas *threads* permitiu a realização da extração do áudio em menor tempo. Nesse caso, as curvas ficam próximas, havendo coincidências entre elas em alguns pontos.

Gráfico 1: Tempo de execução x Número de *threads*/*Chunk* = *1*/Máquina 1



Fonte: Dados da pesquisa

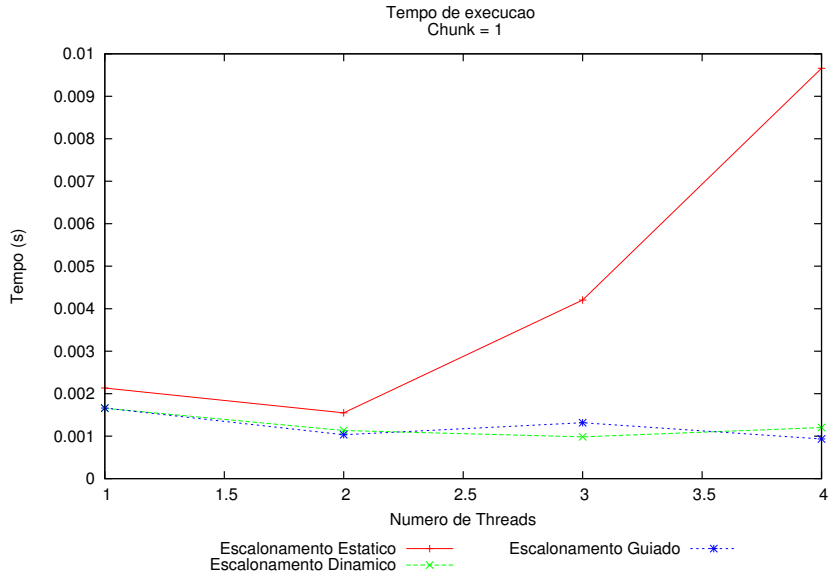
Gráfico 2: Tempo de execução x Número de *threads*/*Chunk* = 32/Máquina 1



Fonte: Dados da pesquisa

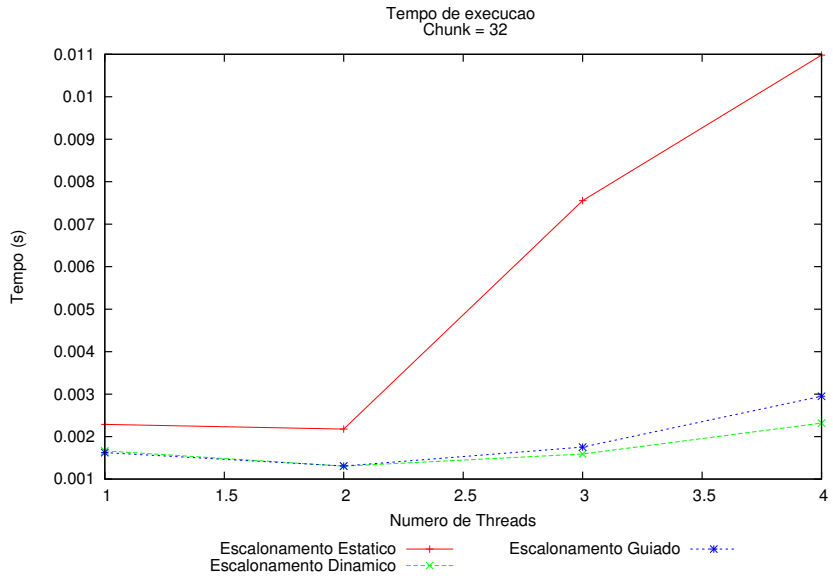
Em relação à Máquina 2, os gráficos 3 e 4 mostram que os escalonamentos guiado e dinâmico possibilitaram a execução do ASR em menor tempo.

Gráfico 3: Tempo de execução x Número de *threads*/*Chunk* = 1/Máquina 2



Fonte: Dados da pesquisa

Gráfico 4: Tempo de execução x Número de *threads*/*Chunk* = 32/Máquina 2



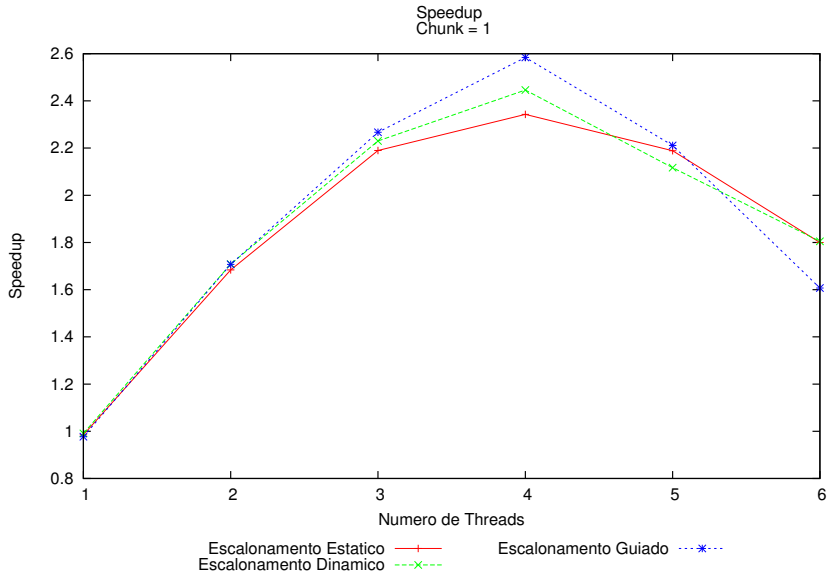
Fonte: Dados da pesquisa

6.1.2 Speedup

Os Gráficos a seguir mostram os resultados do cálculo do *speedup* para Máquina 1 e Máquina 2. Em relação à Máquina 1, conforme demonstrado no Gráfico 5, o escalonamento guiado obteve *speedup* de aproximadamente 2,6 na execução do sistema utilizando-se 4 *threads* e *chunk* = 1. Por outro lado, para *chunk* = 32 o maior *speedup* foi alcançado

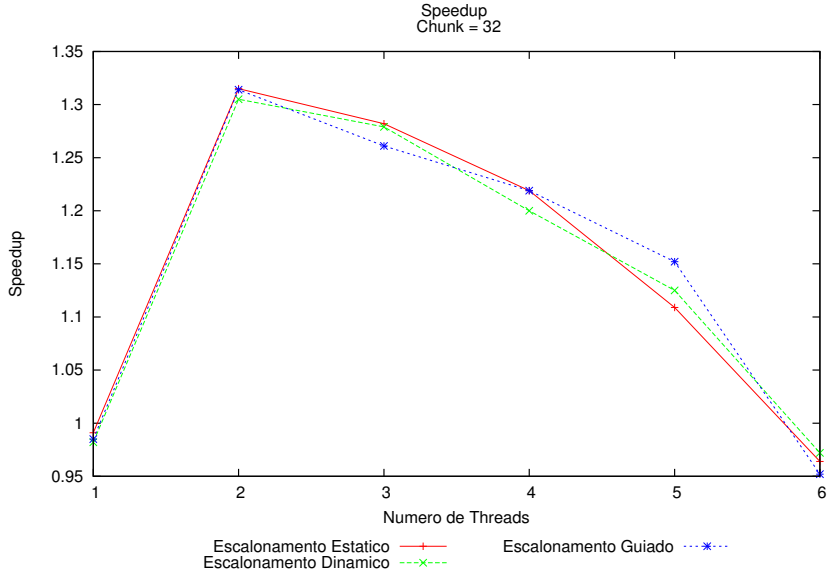
na execução com 2 *threads*, sendo os valores próximos para os três escalonamentos.

Gráfico 5: Speedup x Número de *threads*/*Chunk* = 1/Máquina 1



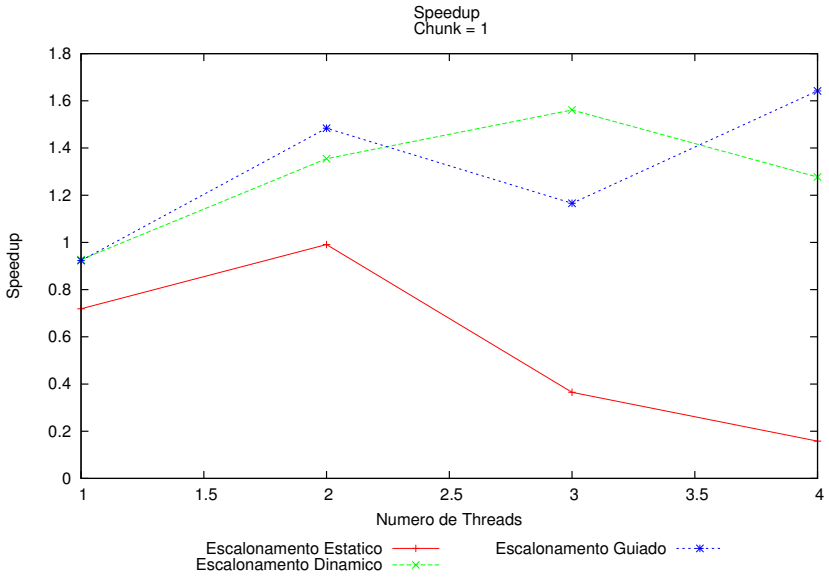
Fonte: Dados da pesquisa

Gráfico 6: Speedup x Número de *threads*/*Chunk* = 32/Máquina 1



Fonte: Dados da pesquisa

Gráfico 7: Speedup x Número de *threads*/*Chunk* = 1/Máquina 2

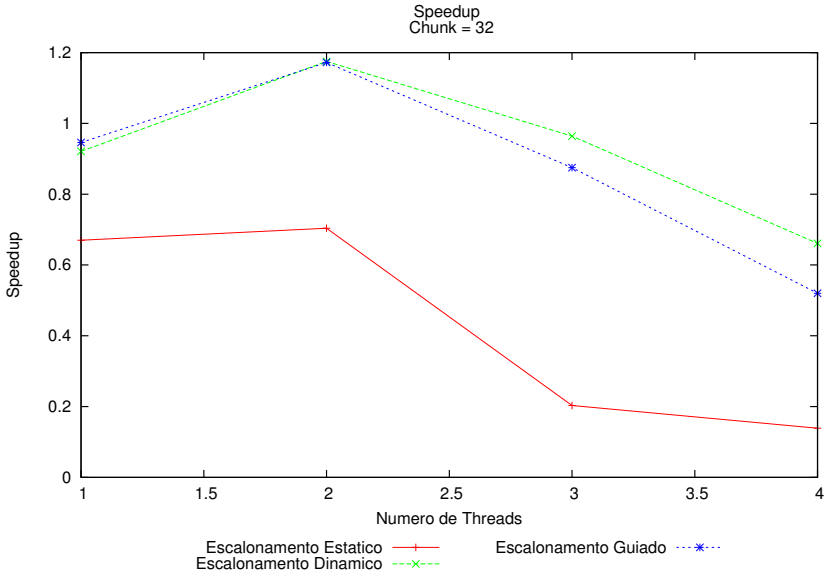


Fonte: Dados da pesquisa

Em relação à máquina 2, alcançou-se *speedup* em torno de 1,5 executando-se o programa com 2 *threads*, utilizando-se escalonamento dinâmico ou guiado. O escalonamento estático apresentou valores não significativos para esse tipo de *hardware*.

Percebe-se que como o extrator de atributos é desbalanceado e possui poucos blocos de entrada, o aumento do *chunk* reduz o ganho de desempenho, principalmente para a Máquina 1, mesmo com a utilização de políticas de escalonamento dinâmico.

Gráfico 8: Speedup x Número de *threads*/*Chunk* = 32/Máquina 2



Fonte: Dados da pesquisa

6.2 Resultados quanto ao desempenho de *hardware*

A avaliação do Hardware foi realizada na Máquina 1 utilizando os seguintes parâmetros:

a) Taxa de previsões de desvio condicional corretas;

b) Taxa de ciclos paralisados aguardando recursos;

c) Taxa de falta de *cache*;

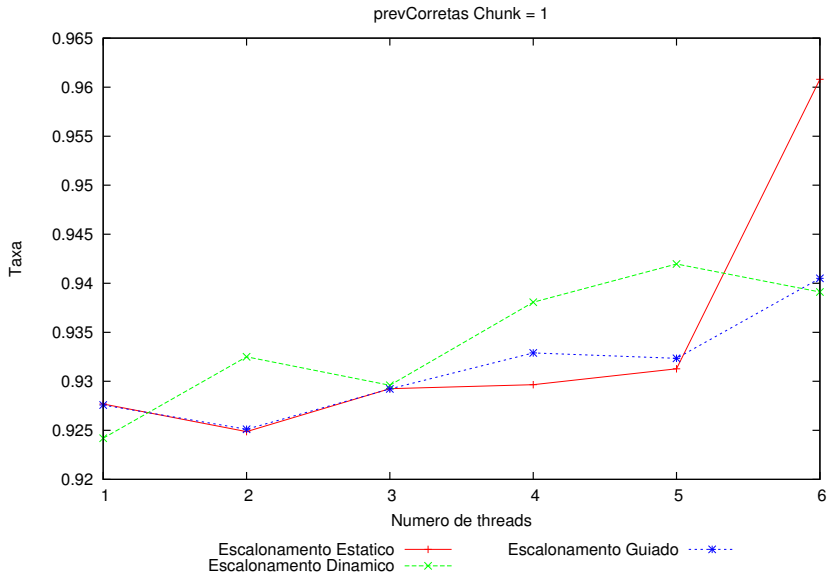
d) Taxa de instruções completadas.

Juntamente com o tempo de execução, estas métricas auxiliam na realização de análise mais detalhada do sistema.

6.2.1 Taxa de previsões de desvio condicional corretas

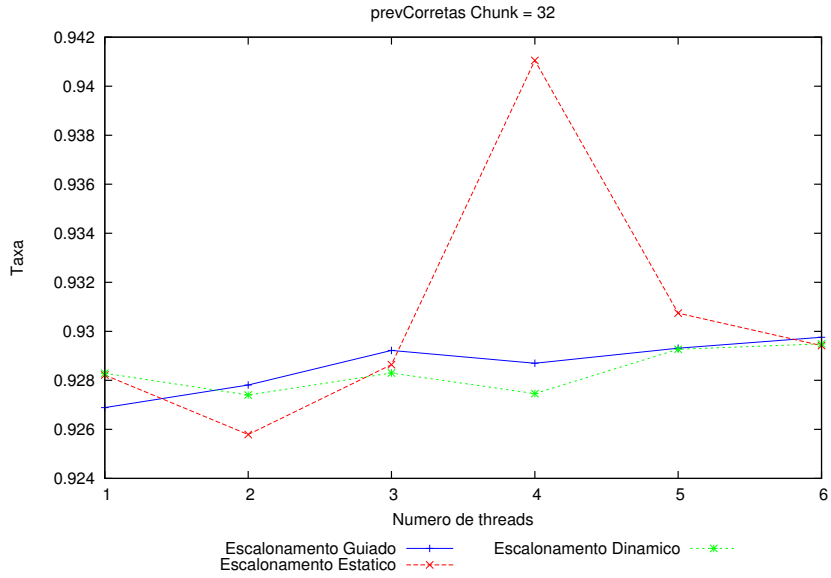
Os Gráficos 9 e 10 apresentam a taxa de previsões realizadas corretamente para tamanhos de *chunk* 1 e 32. Os valores mostram melhora na taxa de previsões corretas à medida em que cresce o número de *threads* utilizadas. Para *chunk* de tamanho 1, apresentado no Gráfico 9, o escalonamento dinâmico mostrou melhores taxas de previsões de desvio corretas utilizando-se 5 *threads*. Já para *chunk* de tamanho 32, conforme Gráfico 9, verificou-se que apesar de a taxa de previsões corretas do escalonamento estático apresentar um salto na execução com 4 *threads*, o escalonamento guiado apresentou melhores valores para a maioria das execuções.

Gráfico 9: Taxa de previsões corretas/ *Chunk* = 1



Fonte: Dados da pesquisa

Gráfico 10: Taxa de previsões corretas/ *Chunk* = 32

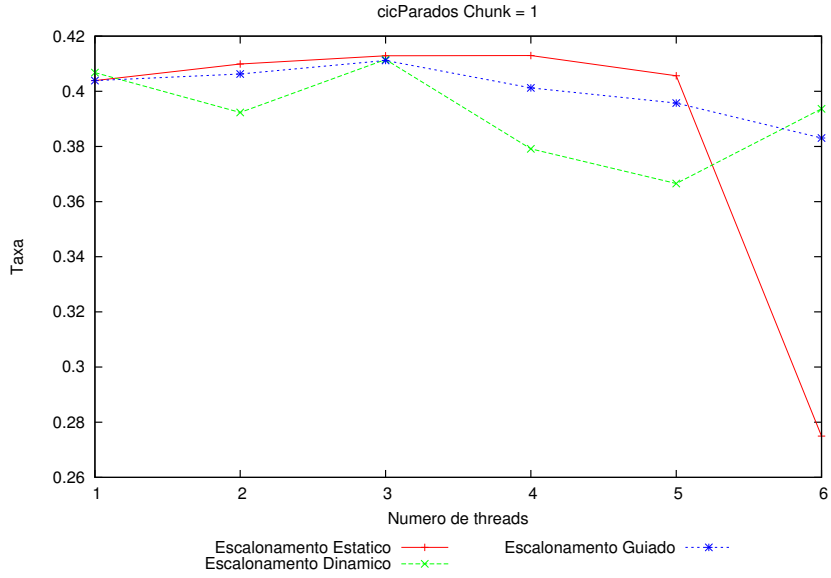


Fonte: Dados da pesquisa

6.2.2 Taxa de ciclos paralisados

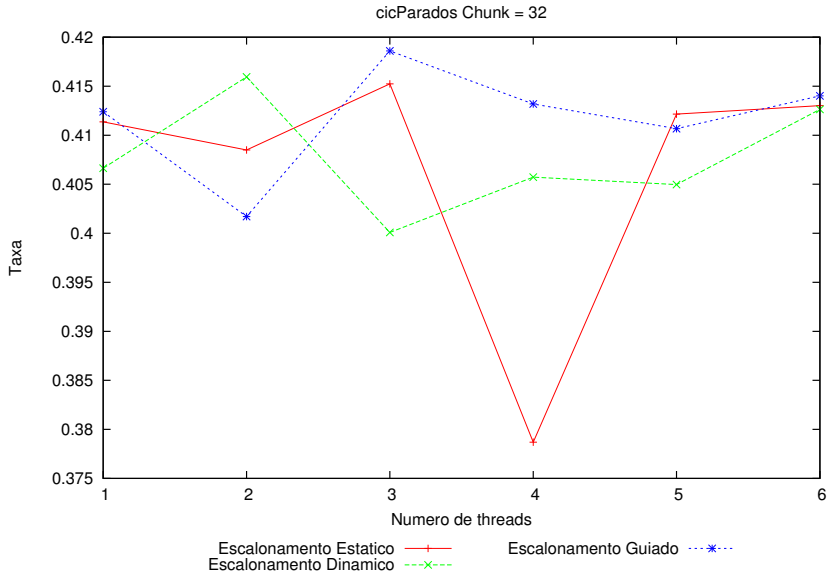
Os resultados da avaliação da taxa de ciclos paralisados aguardando recurso são mostrados nos Gráficos 11 e 12. Na avaliação do sistema utilizando-se *chunk* = 1, percebe-se pequena variação da taxa para os diferentes números de *threads* em comparação à avaliação com *chunk* = 32, exceto na execução com 6 *threads* e escalonamento estático. O escalonamento dinâmico apresentou as menores taxas utilizando-se *chunk* de tamanho 1.

Gráfico 11: Taxa de ciclos paralisados/ *Chunk* = 1



Fonte: Dados da pesquisa

Gráfico 12: Taxa de ciclos paralisados/ *Chunk* = 32

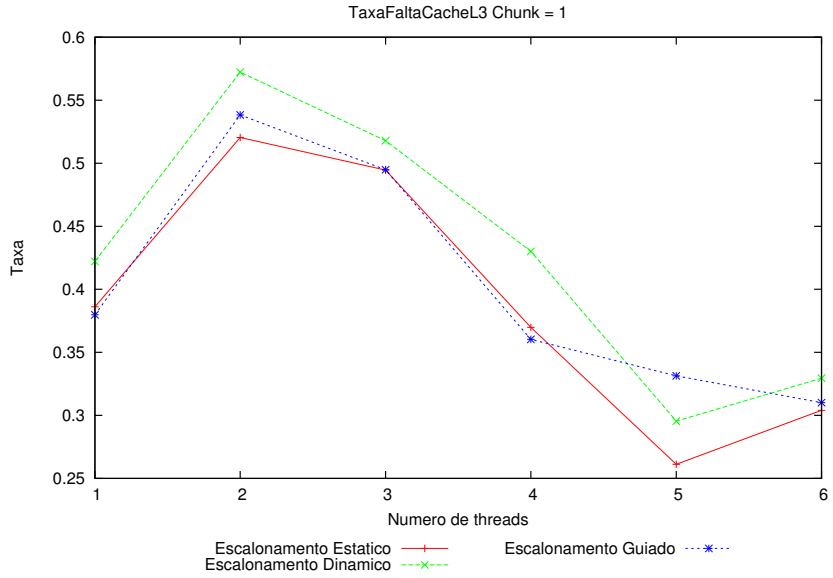


Fonte: Dados da pesquisa

6.2.3 Taxa de falta de cache L3

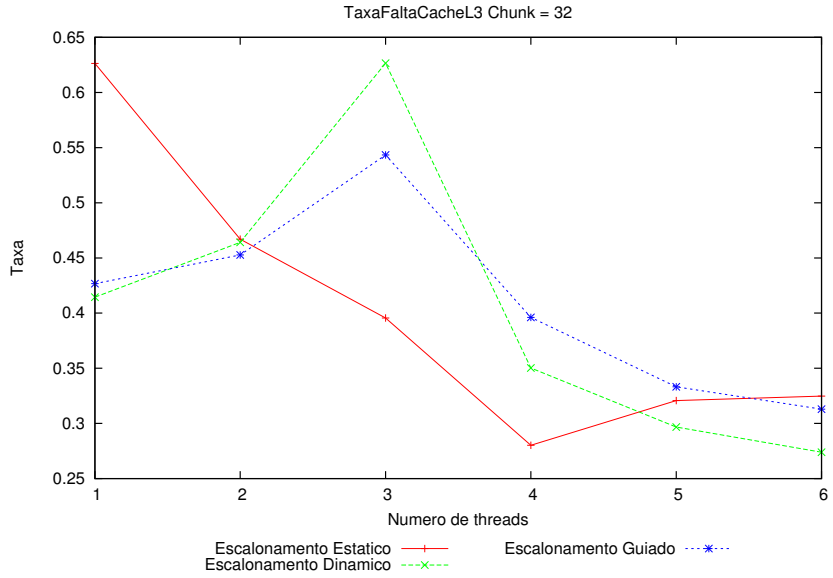
Os resultados da avaliação de taxa de falta de *cache* para *chunks* de tamanho 1 e 32 são apresentados nos Gráficos 13 e 14, respectivamente. Foram verificadas variações em torno de 30%. No primeiro caso, os três escalonamentos apresentaram comportamento semelhante em relação ao número de *threads*. Há um crescimento da taxa entre 1 e 2 *threads* e diminuição entre 2 e 5 *threads*, voltando a crescer na execução com 6 *threads*, com exceção do escalonamento guiado, que continuou em queda. A menor taxa de falta de *cache* para *chunk* = 1 foi observada em ambos os gráficos, na maioria das vezes, utilizando-se escalonamento estático na execução do sistema. Isso se deve ao fato de que com o aumento do número de *threads*, a possibilidade de dados necessários à execução das instruções estarem disponíveis na memória *cache* no momento do acesso da *thread* é maior, ocorrendo diminuição na taxa de falta de cache.

Gráfico 13: Taxa de falta de *cache*/*Chunk* = 1



Fonte: Dados da pesquisa

Gráfico 14: Taxa de falta de *cache*/*Chunk* = 32



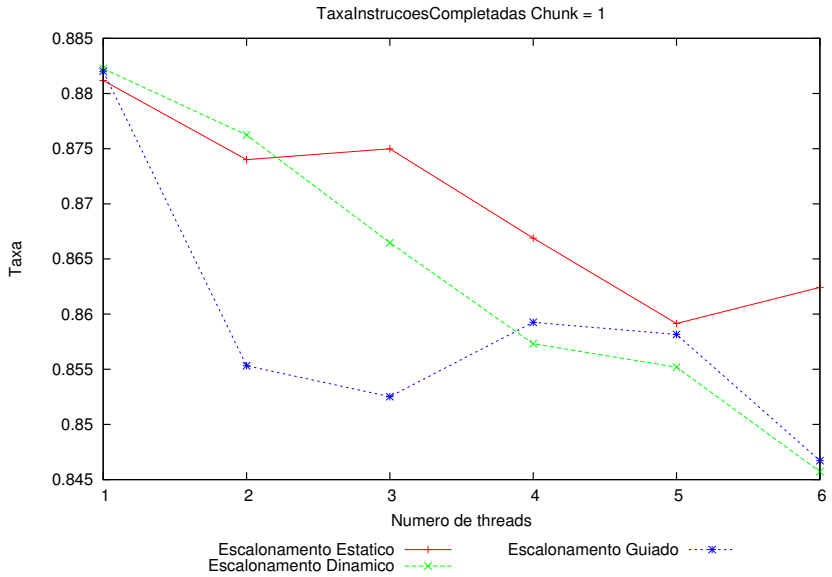
Fonte: Dados da pesquisa

6.2.4 Taxa de instruções completadas

Os resultados para o teste de instruções completadas para *chunk* de tamanho 1 e 32 são apresentados nos Gráficos 15 e 16, respectivamente. Percebe-se a tendência na diminuição da taxa de instruções completadas conforme aumenta-se o número de *thre-*

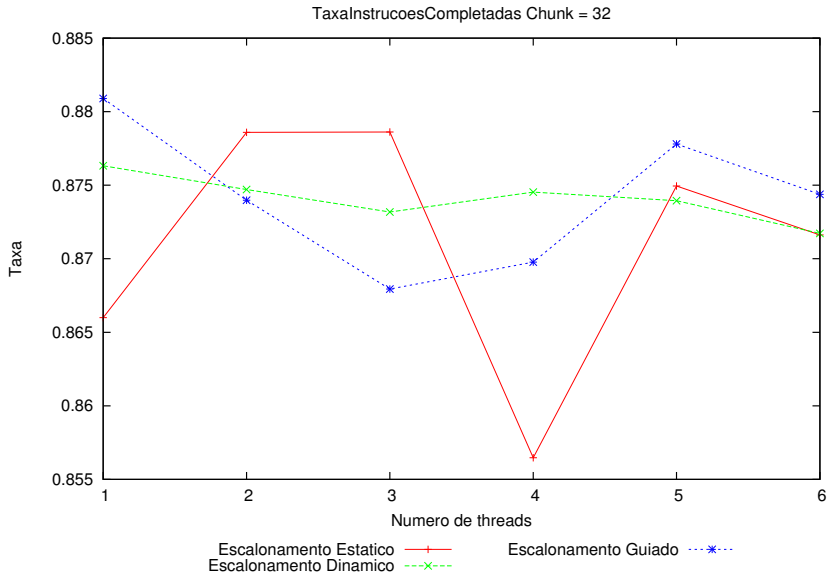
ads, sendo os escalonamentos dinâmico e estático os que apresentam as maiores taxas de instruções completadas.

Gráfico 15: Taxa de instruções completadas/ *Chunk* = 1



Fonte: Dados da pesquisa

Gráfico 16: Taxa de instruções completadas/ *Chunk* = 32



Fonte: Dados da pesquisa

6.3 Análise dos resultados

Em relação ao tempo de execução, para a Máquina 1, observou-se que houve pouca variação no tempo de execução da extração de atributos do ASR, relacionando-se os três tipos de escalonamento. Ainda assim, para tamanho de *chunk* = 1, o escalonamento guiado foi o que apresentou menor tempo, na maioria das vezes. Na avaliação da taxa de falta de *cache*, foi o tipo de escalonamento que apresentou os menores valores, tanto para *chunk* de tamanho 1 quanto 32. Já para *chunk* = 32, os três tipos de escalonamento obtiveram resultados semelhantes em termos de tempo de execução. Já para a Máquina 2, os escalonamentos guiado e dinâmico utilizando-se 2 *threads* possibilitaram a extração de atributos em menor tempo, tanto para *chunk* = 1 quanto para *chunk* = 32.

Os valores de *speedup* apresentaram picos na execução do sistema com 2 e 4 *threads* na máquina 1, utilizando-se *chunk* = 1 e *chunk* = 32, respectivamente. Aumentando-se o número de *threads*, o *speedup* tende a diminuir. Na execução do sistema na Máquina 2, alcançou-se maior valor de *speedup* na utilização de 2 *threads*, acima disso, o valores decrescem.

Conforme apresentado na seção 6.2, houve uma tendência no aumento da taxa de previsões corretas de desvio condicional ao se aumentar o número de *threads*, principalmente para os escalonamentos dinâmico e guiado. Já a taxa de instruções completadas teve tendência em diminuir, assim como a taxa de falta de *cache*. Percebe-se que as características do *hardware* avaliadas não estão relacionadas ao tempo de execução.

Desta forma, a utilização do escalonamento guiado em uma máquina de 6 núcleos e 4 *threads* possibilitou a extração de atributos em menor tempo, se comparado aos outros tipos de escalonamento. Já para a máquina com 2 núcleos, tanto o escalonamento dinâmico como o guiado tiveram desempenho próximo, destacando-se a execução do sistema com 2 *threads* como recomendável.

7 CONCLUSÃO

Neste trabalho realizou-se a implementação paralela da etapa de extração de atributos e avaliação de um sistema de reconhecimento automático de fala baseado em HMM para palavras isoladas e vocabulário reduzido (até 99 palavras). O sistema foi implementado na linguagem C, juntamente com a API de programação paralela OpenMP. Foram avaliados o tempo de execução e o desempenho de *hardware*, através da ferramenta PAPI, em ambiente Linux. Os testes não consideraram o tempo de leitura e escrita de arquivos pelo fato de não apresentar impacto no desempenho do sistema.

Os resultados mostram que o escalonamento guiado proporcionou redução em torno de 63% no tempo de execução paralelo em relação à implementação sequencial da extração de atributos, em uma máquina com processador de 6 núcleos, utilizando 4 *threads*. Já para uma máquina de 2 núcleos a execução com 2 *threads* reduziu o tempo de execução em torno de 15% , tanto com a utilização do escalonamento guiado quanto do escalonamento dinâmico.

Em relação ao objetivo geral do trabalho, conclui-se que foi alcançado, uma vez que foi executada a avaliação e foram observados ganhos no desempenho do sistema. Além disso, os conceitos abordados contribuem para facilitar a realização de trabalhos futuros ao relacionar duas áreas distintas, tais como o reconhecimento automático de fala e computação paralela. Por ser uma etapa comum a sistemas de processamento de voz, este estudo pode ser estendido a outras aplicações tais como recuperação de informação em áudio, sistemas de análise utilizados na área de biologia para análise de sons em animais, dentre outros. Além disso, a utilização dos contadores de *hardware* proporcionados pela ferramenta PAPI possibilitam uma análise mais completa do código, contribuindo para enriquecimento do trabalho.

7.1 Trabalhos futuros

Uma proposta para trabalhos futuros é realizar a avaliação de sistemas de reconhecimento de fala com vocabulário maior que 99 palavras, para verificar a influência do tamanho do vocabulário no desempenho do sistema paralelo. Além disso, buscar combinação ótima entre configurações dos modelos objetivando manter a taxa de reconhecimento e tempo de processamento.

Outra proposta é estender esta avaliação a sistemas de reconhecimento de fala contínua, utilizando Modelos Ocultos de Marcov. Isso exige esforço computacional para aplicações em tempo real.

Referências Bibliográficas

AL, P. M. et. *Performance Application Programming Interface*. 2015. PAPI. Disponível em: <http://icl.cs.utk.edu/papi/overview/>.

ALENCAR, V. F. S. *Atributos e Domínios de Interpolação Eficientes em Reconhecimento de Voz Distribuído*. Dissertação (Mestrado) — Pontifícia Universidade Católica do Rio de Janeiro, 2005.

BAHOURA, M.; EZZAIDI, H. Hardware implementation of mfcc feature extraction for respiratory sounds analysis. In: *Systems, Signal Processing and their Applications (WoSSPA), 2013 8th International Workshop on*. [S.l.: s.n.], 2013. p. 226–229.

BUARQUE, C. Roda viva. In: CHICO BUARQUE. Brasil: RGE, 1967.

CHANDRA, R. et al. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers, 2001. (ISBN 1-55860-671-8). Disponível em: <http://icl.cs.utk.edu/papi/overview/>.

DAN, Z.; MONICA, F. A study about mfcc relevance in emotion classification for srol database. In: *Electrical and Electronics Engineering (ISEEE), 2013 4th International Symposium on*. [S.l.: s.n.], 2013. p. 1–4.

DAVIS, S. B.; MERMELSTEIN, P. Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, p. 357–366, 1980.

DEPARTMENT, M. I. L. of the C. U. E. *The Hidden Markov Model Toolkit*. 1993. Disponível em: <http://htk.eng.cam.ac.uk/>.

DIAZ, J.; MUNOZ-CARO, C.; NINO, A. A survey of parallel programming models and tools in the multi and many-core era. *Parallel and Distributed Systems, IEEE Transactions on*, v. 23, n. 8, p. 1369–1386, Aug 2012. ISSN 1045-9219.

FLYNN, M. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, C-21, n. 9, p. 948–960, Sept 1972. ISSN 0018-9340.

HUANG, X.; ACERO, A.; HON, H.-W. *Spoken Language Processing: A Guide to Theory, Algorithm, and System Development*. 1st. ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001. ISBN 0130226165.

KOU, H.; SHANG, W. Parallelized feature extraction and acoustic model training. In: *Digital Signal Processing (DSP), 2014 19th International Conference on*. [S.l.: s.n.], 2014. p. 503–508.

KOU, H. et al. Efficient mfcc feature extraction on graphics processing units. In: *Signal Processing (CIWSP 2013), 2013 Constantinides International Workshop on.* [S.l.: s.n.], 2013. p. 1–4.

LAWSON, A. et al. Survey and evaluation of acoustic features for speaker recognition. In: *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on.* [S.l.: s.n.], 2011. p. 5444–5447. ISSN 1520-6149.

MAJID, M.; MIRZAEI, G.; JAMALI, M. Parallelization of feature extraction techniques on consumer-level multicore system. In: *Electro/Information Technology (EIT), 2012 IEEE International Conference on.* [S.l.: s.n.], 2012. p. 1–4. ISSN 2154-0357.

MAKA, T.; DZIURZANSKI, P. Parallel audio features extraction for sound indexing and retrieval systems. In: *ELMAR, 2013 55th International Symposium.* [S.l.: s.n.], 2013. p. 185–189. ISSN 1334-2630.

RABINER, L. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, v. 77, n. 2, p. 257–286, Feb 1989. ISSN 0018-9219.

RAJASEKARAN, S.; REIF, J. *Handbook of Parallel Computing: Models, Algorithms and Applications (Chapman & Hall/Crc Computer & Information Science Series)*. 1. ed. [S.l.: s.n.], 2008. ISBN 1584886234, 9781584886235.

RAUBER, T.; RUNGER, G. *Parallel Programming for Multicore and Cluster Systems.* 2. ed. [S.l.]: Springer-Verlag Berlin Heidelberg, 2010. (e-ISBN 978-3-642-04818-0, v. 1).

SANTOS, S. *Reconhecimento de voz contínua para o português utilizando modelos de Markov Escondidos.* Dissertação (Mestrado) — Departamento de Engenharia Elétrica - Pontifícia Universidade Católica do Rio de Janeiro, 1997.

WANG, H.; XU, Y.; LI, M. Study on the mfcc similarity-based voice activity detection algorithm. In: *Artificial Intelligence, Management Science and Electronic Commerce (AIMSEC), 2011 2nd International Conference on.* [S.l.: s.n.], 2011. p. 4391–4394.

YOU, K.; LEE, Y.; SUNG, W. Openmp-based parallel implementation of a continuous speech recognizer on a multi-core system. In: *Acoustics, Speech and Signal Processing, 2009. ICASSP 2009. IEEE International Conference on.* [S.l.: s.n.], 2009. p. 621 –624. ISSN 1520-6149.