TSINGHUA SCIENCE AND TECHNOLOGY
ISSN 1007-0214 11/24 pp447–456
DOI: 10.26599/TST.2023.9010018
Volume 29, Number 2, April 2024

Performance of Text-Independent Automatic Speaker Recognition on a Multicore System

Rand Kouatly and Talha Ali Khan*

Abstract: This paper studies a high-speed text-independent Automatic Speaker Recognition (ASR) algorithm based on a multicore system's Gaussian Mixture Model (GMM). The high speech is achieved using parallel implementation of the feature's extraction and aggregation methods during training and testing procedures. Shared memory parallel programming techniques using both OpenMP and PThreads libraries are developed to accelerate the code and improve the performance of the ASR algorithm. The experimental results show speed-up improvements of around 3.2 on a personal laptop with Intel i5-6300HQ (2.3 GHz, four cores without hyper-threading, and 8 GB of RAM). In addition, a remarkable 100% speaker recognition accuracy is achieved.

Key words: Automatic Speaker Recognition (ASR); Gaussian Mixture Model (GMM); shared memory parallel programming; PThreads; OpenMP

1 Introduction

Human interaction with computers has become more pervasive and essential in financial services, security, and information retrieval from speech databases. Machines' ability to recognize a speaker with high accuracy and speed is a challenge nowadays. The problem is even more challenging when the speech signal used for constructing the features database (training stage) differs from the speech signal used in testing or the real-time application (testing stage). This is called text-independent Automatic Speaker Recognition (ASR). To achieve high recognition accuracy, many training speech signals must be used to construct a speaker database. A big problem arises due to the high complexity of both phases: (1) training and (2) the search for the best candidate in the testing phase. One of the methods used to increase the recognition accuracy, which has been used for a long time, is Gaussian Mixture Model (GMM)[1, 2]. Several methods have been proposed to improve GMM accuracy in text-independent ASR[2–5]. Still, these studies are not concerned with the computation of the features, the complexity of GMM classification techniques, or data size (number and size of the speech signals) used in the training and testing phases. This turns speaker recognition into a computationally demanding problem[6, 7].

Ganjeizadeh et al.[6] studied the best combination between the frame size and the number of GMM parameters to achieve less computational burden. Unfortunately, no new methods were introduced in that work. Petracca et al.[8] introduced a new idea by using speech bitstream information of compressed speaker signals instead of computing the features to reduce the computational burden. Their approach shows inferior results in terms of recognition accuracy when the used speech signal is less than 15 s. In Refs. [9–11], the GMM is implemented in Python and on a Graphic Processing Unit (GPU). The results are poor recognition when quick changes exist in the speakers in the conversation. In this work, we parallelize the text-independent speaker recognition algorithm on a multicore system using

• Rand Kouatly and Talha Ali Khan are with Faculty of Tech and Software Engineering, University of Europe for Applied Sciences, Potsdam 14469, Germany. E-mail: rand.kouatly@ue-germany.de; talhaali.khan@ue-germany.de.
* To whom correspondence should be addressed.
Manuscript received: 2022-09-26; revised: 2023-03-10; accepted: 2023-03-18

© The author(s) 2024. The articles published in this open access journal are distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/).

the C language and the shared memory programming standards OpenMP[12] and PThreads[13]. This paper is organized as follows: Section 2 shows an overview of the speaker recognition system, highlighting the high computational burden of getting the speaker features, and how to deal with it. Section 3 shows an overview of the GMM, focusing on its workload. An overview of the parallel implementation of the ASR system is presented in Section 4. Simulation results on an i5-6300HQ 2.3 GHz with 4 GB of RAM are shown in Section 5, and the main conclusions and future work are shown in Section 6.

2 Speaker Recondition System

Figure 1 shows training and testing phases of a speaker verification system. A speaker verification system is based on two stages: Training and testing. The training phase is used to generate the models of the speakers, which are used to identify speakers in the test phase. Figure 1 shows the components of the speaker recognition system, where the upper part represents the training.

Each phase consists of several serial independent modules. In both phases, the first step consists of extracting the speaker parameters from the speech signal to obtain a suitable representation for the second phase (feature aggregation). This step will be described in detail in Section 3. The first step aims to transform the speech signal into a set of speech features to remove redundancy and reduce the data size, favouring statistical modelling. This is achieved by computing the Mel Frequency Cepstral Coefficients (MFCC) representation[14]. Figure 2 shows the block diagram for the MFCC computation.

In MFCC, the speech signal is first divided into 20–30 ms vectors. The speech vector can be overlapped to improve recognition[2]. The per-emphasized method is

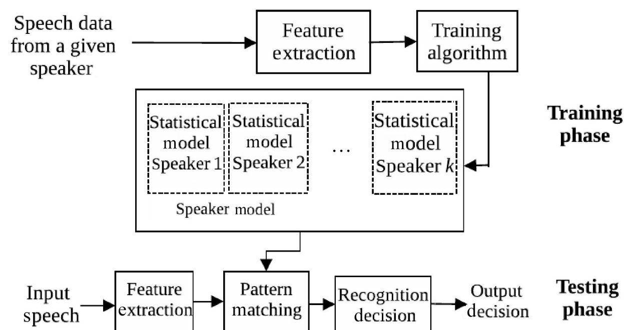


Fig. 1 Training and testing phases of a speaker verification system.

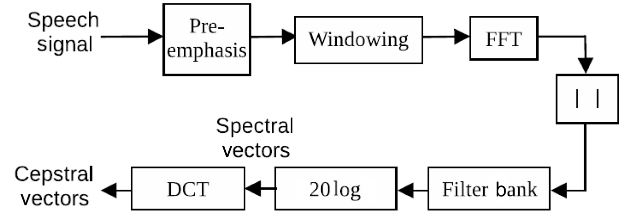


Fig. 2 Computation of MFCC vectors[7].

used to eliminate the high frequencies of the spectrum and can be applied to signal x using the first-order filter as follows[15, 16]:

$$x_P(t) = x(t) - \alpha x(t-1) \tag{1}$$

where α is the filter coefficient in the range [0.95, 0.98][17], the pre-emphasized signal is then windowed using Hanning window(s) to improve the spectral representation of the speech vector[18]. Once the speech signal has been windowed and pre-emphasized, the Fast Fourier Transform (FFT) is calculated[15]. Then, the modulus of the FFT vector is extracted (dended by | | in Fig. 2). A filter bank filters the returned vector to smooth the signal spectrum because only the envelope of the spectrum is of interest; a typical of 40 filters are used[16]. A filter bank contains a series of bandpass frequency filters multiplied by the spectrum to get an average value in a particular frequency band. Several filter banks have been used in Refs. [15, 16]. The Bark/Mel is one of the most frequently used to convert between the frequency f (in Hz) and $f_{\rm Mel}$[14],

$$f_{\rm Mel} = 1000\frac{\log(1+\frac{f}{1000})}{\log 2} \tag{2}$$

The $f_{\rm Mel}$ scale aims to mimic the non-linear human ear perception of sound by being more discriminative at lower frequencies and less discriminating at higher frequencies. Next, the log of this spectral envelope is calculated, and each coefficient is multiplied by 20 to obtain the spectral envelope in dB. Finally, the Discrete Cosine Transform (DCT) is usually applied to the spectral vectors in speech processing, resulting in the Cepstral coefficients[19],

$$c_n = \sum_{k=1}^{N} x^k \cos\left(n\left(k-\frac{1}{2}\right)\frac{\pi}{2}\right),\ n = 1, 2, \ldots, L \tag{3}$$

where N is the number of previously calculated log-spectral coefficients, L is the number of the Cepstral coefficients we want to obtain, and x is the speech samples at the k-th frame[20].

3 GMM

After extracting features for each speech frame, an

aggregation method is usually performed to obtain the summary values for each speaker. Several ways exist for feature aggregation. One is the GMM[5, 21], a GMM comprises a finite weight of the sum of M of multivariate Gaussian components densities. A GMM is characterized by its probability density function[22] as follows:

$$p(x|\lambda)=\sum_{i=1}^{M}\omega_i g(x|\mu_i,\Sigma_i) \tag{4}$$

where x is a d-dimensional continues-valued speech features, ω_i (i=1, 2, ..., M) are the mixture weights, and $g(x|\mu_i,\Sigma_i)$ (i=1, 2, ..., M) are the components of Gaussian densities. For each component, the variance is a d variance of the Gaussian density function[22],

$$g(x|\mu_i,\Sigma_i)=\frac{1}{(2\pi)^{\frac{d}{2}}|\Sigma_i|^{\frac{1}{2}}}\exp\left[-\frac{1}{2}(x-\mu_i)^{\mathrm{T}}\Sigma_i^{\mathrm{T}}(x-\mu_i)\right] \tag{5}$$

where μ_i is the d-dimensional mean vector, Σ_i is a covariance matrix with $d\times d$ dimension, and the mixture weight satisfies the constraint by $\sum_{i=1}^{M}\omega_i=1$[23]. The mean vectors, covariance matrices, and mixture weights from all component densities parameterize the complete Gaussian mixture model. These parameters are collectively represented by the notation[22],

$$\lambda=\{\omega_i,\mu_i,\Sigma_i\},\quad i=1,2,\ldots,M \tag{6}$$

In the testing phase, the N speakers database is represented by a GMM. The tested speaker is identified as that who has the maximum posterior probability density function (see Eq. (7) in the following) among the population. This is mainly done using the Expectation Maximization (EM) algorithm for GMM[22, 24]. For a sequence of T training vectors $X=\{x_1,x_2,\ldots,x_T\}$, the GMM likelihood can be given by Ref. [21],

$$p(X|\lambda)=\prod_{t=1}^{T}p(x_t|\lambda) \tag{7}$$

The parameter estimation can be obtained iteratively by starting from an initial model of λ (K-mean a very good choice initial value), then processed to update λ iteratively until convergence is detected (significant change of the log-likelihood value, see Eq. (12)), or the number of maximum of iteration is reached. In the EM iteration, Eqs. (8)–(10) in the following are used to estimate the mixture weights, means, and variances, respectively[24]:

$$\bar{\omega}_i=\frac{1}{T}\sum_{t=1}^{T}\Pr(i|x_t,\lambda) \tag{8}$$

$$\bar{\mu}_i=\frac{\sum_{t=1}^{T}\Pr(i|x_t,\lambda)x_t}{\sum_{t=1}^{T}\Pr(i|x_t,\lambda)} \tag{9}$$

$$\bar{\sigma}_i^2=\frac{\sum_{t=1}^{T}\Pr(i|x_t,\lambda)x_t^2}{\sum_{t=1}^{T}\Pr(i|x_t,\lambda)}-\bar{\mu}_i^2 \tag{10}$$

where the posterior probability for each component i is given in the following:

$$\Pr(i|x_t,\lambda)=\frac{\omega_i g(x_t|\mu_i,\Sigma_i)}{\sum_{k=1}^{M}\omega_k g(x_t|\mu_i,\Sigma_i)} \tag{11}$$

In the M-step, the membership for data points is changed slightly, and the values of λ and mixture weights are recomputed. The iterations (E-step and M-step) are repeated until the maximum log-likelihood[21] is achieved, or the maximum number of iterations M is reached,

$$E\{\log P(X|\lambda)\}=\sum_{t=1}^{T}\sum_{k=1}^{M}\Pr(k|x_k,\lambda)\left\{\log\omega_k+\log g(x_t|\mu_k,\Sigma_k)\right\} \tag{12}$$

The computational burden of the ASR algorithm is very high in both the feature extraction and aggregation stages. This affects both the speed of the speaker identification system due to high processing time and also to the accuracy of the identification. For the latter, the designer tends to reduce the number of features and increase the speech frame size to reduce the burden of the ASR system in most real-time applications.

4 Parallel Implementation

Parallelism is the type of computation used to accelerate many hard-to-solve problems. We are interested in shared-memory systems because it is easier to develop parallel algorithms on them from the programmer's point of view[12, 25]. Communication and data replication often lead to additional high overheads in distributed memory systems but offer better scalability than shared-memory ones. Shared-memory programming has to be handled with care. For instance, many possible interleaving of threads might exist, creating errors in the results that are difficult to detect[26]. Instead of using threads directly, one can use simple constructs that indicate which parts

of the program are safe to run in parallel. On the one hand, languages, such as OpenMP[27], Intel Threading Building Blocks[28], Habanero[29], and X10[30], are examples of libraries that allow the programmer to leave to the run-time scheduler essential decisions like how to assign work to threads and how to perform the load balancing among them[31]. On the other hand, PThreads provide more control to the programmer with a lower programming level than in the libraries, as mentioned above[13].

Here, we focus on both PThreads and OpenMP. PThreads programming interface is typically accessed via a run-time library and operating system calls. Any ANSI/ISO-C conforming compiler may compile programs with the PThreads library. OpenMP extends programming C with directives to make explicit parallelism and data privacy. OpenMP annotations change the semantics of loops and data persistence. As in all programming models, special care is needed because OpenMP annotations can assert incorrect program transformations, manifesting as race conditions or deadlocks[32].

The ASR system consists of sequential stages, i.e., each step depends on the results of the previous stages. So, our methodology is based on parallelizing each stage independently, moving the results to the next stage, and so on. Parallelizing ASR aims to increase the speed without harming the algorithm's accuracy. Figure 3 shows the parallelization principle of the ASR stages.

We have created a benchmark suite to evaluate the parallel performance of ASRs, as was done in Ref. [33]. The suite contains two main benchmarks. The first benchmark corresponds to the MFCC evaluation (see Section 2) and the second to the GMM one (see Section 3). Each benchmark has several sub-stages, as shown in Table 1. In Table 1, N represents the

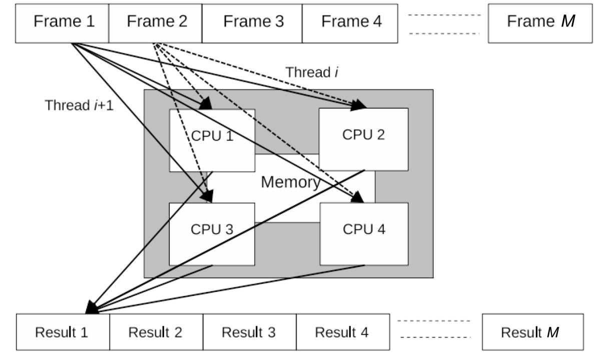


Fig. 3 CPU parallelization principle of GMM and MFCC stages.

number of samples in each frame, T is the number of Cepstral coefficients computed for each frame, M is the number of Gaussian components or clusters, and K is the maximum number of iterations (see Section 3). For the simulation benchmark the maximum number of iteration $K = 10\,000$ is used as the threshold if the maximum likelihood is not reached, $M = 12$ is the number of clusters selected to equal the number of MCFF parameters for each speech frame. P is the order of the coefficients in each frame. The number of Lines Of Code (LOCs) for each algorithm is used in each benchmark. In addition, sequential, PThreads, and OpenMP algorithms have been developed for each benchmark. The following sections show that PThreads and OpenMP variants exploit the exact parallelism.

4.1 Parallelization of the feature extraction

The parallelization of sub-stages of MFCC (preEmphasis, fft, fhank, energy, and dct) has been done by a static distribution of the input data among threads based on parallel loops. After all the threads have finished, their results are gathered. The time waiting for all threads to finish is the main

Table 1 ASR simulation benchmark.

Method	Stage	Sub-stage	Complexity order	Number of LOCs
MFCC	MFCC	–	$O(N^2P)$	1631
	Pre-emphasis and windowing	preEmphasis	$O(N)$	22
	FFT	fft	$O(N\log N)$	730
	Filtering	fbank	$O(N)$	45
	Energy	energy	$O(N)$	10
	DCT	dct	$O(N^2)$	258
GMM	GMM	–	$O(KT^3)$	1609
	Initialization	gmmInt	$O(T^2)$	730
	E-step training	gmmETrain	$O(T^2KM))\simeq O(KT^3)$	119
	M-step and log-likelihood computation	gmmMTrain	$O(T^2PM))\simeq O(KT^3)$	131
	Classification and log-likelihood computation	gmmClassifier	$O(T^2\ KP)\ )\simeq O(KT^2)$	159

limitation of the efficiency.

Algorithm 1 shows the pseudo-code of the parallelization of the MFCC benchmark. A buffer is created with different frames extracted from the speech vector, and the sequence of sub-stages for MFCC in Table 1 is applied sequentially for each frame until the end of the speech signal(s).

The parallel algorithms for each sub-stage are shown in Algorithms 2 to 6, as a result of the MFCC, the extracted features are stored for the next GMM benchmark.

Algorithm 1 MFCC ($S, \alpha, \varepsilon, L$, NTh)

```
Require:
    S: speech frame,
    C_n : output coefficients vector,
    α: pre-emphasis coefficient,
    ε: energy coefficient,
    L: number of Cepstral coefficients,
    NTh: number of threads.
1: for i ← 1 to M do
2:    Eng ← energy (S_i, ε, NTh);              → See Algorithm 2
3:    Sp ← preEmphasis (S_i, Eng, NTh);        → See Algorithm 3
4:    fft (Sp, S_i, N, NTh);                   → See Algorithm 4
5:    S ← fbank (S_i, Eng, NTh);               → See Algorithm 5
6:    C_n ← dct (S_i, L, NTh);                 → See Algorithm 6
7: return C_n
```

Algorithm 2 energy (S, ε, NTh)

```
Require:
   S: speech frame,
   ε: energy coefficient,
1: for all threads i ← 0 to N/NTh do
2:    ε ← ε + S_i × S_i;
3:    reduction (ε);
4: if ε > 0 then
5:      return log (ε);
6: else
7:      return 0
```

Algorithm 3 preEmphasis (S, α, NTh)

```
Require:
    S: speech frame,
    α: pre-emphasis coefficient,
    NTh: number of threads,
    N: frame length.
1: for all threads i ← 0 to N/NTh do
2:    S_pi ← S_i + S_{i−1} × α                 → Pre-emphasis part
3: for all threads i ← 0 to N/NTh do
4:    S_pi ← S_pi × [0.54−0.46 × cos(2πi/(N− 1))];
                                               → Hamming window part
5:    reduction (S_p);
6: return S_p
```

Algorithm 4 fft (x, y, n, NTh)

```
Require:
    x: input frame,
    y: output frame,
    n: vector length,
    NTh: number of threads.
1: l ← 2^n;                 → Length of x [ ] is the power of 2
2: y ← bit reverse-permutation (x);
3: for all threads i ← 1 to (n/2)/NTh do
4:       ω_d ← e^{2πi/d}
5: for j ← 0 to log (n) do
6:       d ← 2^j;
7:       ω ← 1;
8:       for all threads k←0 to (d/2 −1)/NTh do
9:          for all threads m ← k to (n − i)/NTh step d do
10:              t ← ω × y[m + d/2];
11:              x ← y [k];
12:              y [k] ← x + t;
13:              y [k + d/2] ← x − t;
14: ω ← ω × ω_d;
15: return
```

Algorithm 5 fbank (x, n, f_s, ε, NTh)

```
Require:
   S: speech frame,
   α: pre-emphasis coefficient,
   ε: eneray coefficients,
   n: vector length,
   f_s: sampling frequency,
   NTh: number of threads.
1: for all threads i ← 0 to n/NTh do
2:    f_Mel ← α log(1 + kf_s/b);
3:    count ← 0;
4: for k ← 0 to n/2 do
5:    while f_Mel < kf_s/n and count ⩽ n do
6:       count ← count+1;
7:       p_Mel ← count;
8: for all threads k ← 1 to (n/2) NTh do
9:       j ← p_Mel;
10:      w ← (f_Mel j −kf_s/n)/f_Mel0;
11: for all threads k ← 1 to n/NTh do
12:     j = f_Mel;
13:     if j > 0 then
14:         y_j ← y_j + x × w;
15:     if j ⩽ n then
16:         y_{j+1} ← y_{j+1} + x(1 − w);
17: for all threads k ←1 to n/NTh do
18:       if y < ε then
19:          y ← ε;
20:          y ← log(y);
21: return y
```

In Algorithms 2 and 3, the speech frame is divided into equal-sized chunks processed in parallel by threads (see Algorithm 2 Line 1 and Algorithm 3 Lines 1 and

Algorithm 6 dct (x, n, NTh)

```
Require:
   x: input frame,
   y : output frame,
   n: vector length,
   Nth: number of threads.
1: if n = 1 then
2:    y ← x;
3: else
4:    m←2^n;                         → Length of x[ ] power of 2
5:    x_1 ← bit reverse-permutation (x);
6:    for all threads i ← 1 to n/NTh do
7:       ω_d ← e^{2πi/d};
8:    fft (x_1, r, n, NTh);           → See Algorithm 4
9:    for all threads k←0 to m/NTh do
10:      y ← r × ω_d;
11:   for k ← 2 to n do
12:      y ← y × √2
13: return y
```

3). The results generated from each thread are gathered as soon as possible, considering the dependencies in the loop.

Algorithm 4 shows the parallel implementation of the Radix 2 FFT based on the iterative method taken from Refs. [34, 35]. The first step performs the permutation of the input vector. Each element $x[i]$ is copied to $y[j]$, where j is the index found by reversing the bits of $y[i]$[35]. Then the parallel computation of the cos and sin table is performed. The main loop of the function has log(n) iterations. During each iteration, each task computes its new value of $y[k]$ from the previous values of $y[k]$ and either $y[k+m/2]$ or $y[k-m/2]$[35]. Most of the computational burden is in the inner two loops that perform the complex numbers' required multiplication, addition, and subtraction.

Algorithm 5 shows the parallel implementation of the Mel filter bank based on signal processing tool Kit SPTK[11].

In Algorithm 5, the first step is computing the filter Mel frequencies and storing them in vector f_{Mel}, where a and b are constants (see Eq. (2)). The second step is to compute filter points p_{Mel} where these points are spaced using Mel frequencies f_{Mel}. Using the values calculated in the first and second steps, the weighting filter vector is computed using a parallel loop where the index j is shared between loops. In the filtering section, there exist dependencies in the loop parameters. In OpenMP, multiple code versions have been created and passed for different thread capabilities using the simd techniques[36, 37]. The last step is to perform the log of the filtered output to prepare the spectral vector (see Fig. 2). Algorithm 6 shows the parallel implementation of the DCT[38]. The DCT uses the FFT in Algorithm 4. The parallelization is primarily performed in the part of the DCT loop where the real part and imagery part of the input are added and multiplied by the cosine and sine value (describe variables in Algorithm 6).

4.2 Parallelization of the GMM procedure

In the following, we describe the parallelization of the different sub-stages of the GMM procedure. They suffer from the same efficiency limitation as the one in feature extraction. The computational workload is assigned to the available idle threads during the process. Algorithm 7 shows the pseudo-code of the GMM benchmark.

The GMM benchmark starts initiating the GMMs and copying the model data. Then it computes the GMM weighting density function (see Algorithm 8) in the E-step and changes the parameters to calculate the log-likelihood during the M-step (see Algorithm 9). These steps are repeated until a slight difference among

Algorithm 7 GMM (C_n, n, NTh)

```
Require:
   C_n : MFCC vector,
   n : vector length,
   NTh: number of threads.
1: for i ← 0 to M do
2:    for i ← 1 to max do
3:    gmmInt ();
4:    w_ik ← gmmETrain (C_n, n, NTh);   →see Algorithm 8
5:    llh_{t,i} ← gmmMTrain (w_ik, C_n, n, NTh);
                                         →see Algorithm 9
6:    if llh_{t,i} ≃ llh_{t−i,i} then
7:       return llh_{t,i};
8: return 0
```

Algorithm 8 gmmETrain (C_n, n, NTh)

```
Require:
   C_n: MFCC vector,
   n: vector length,
   NTh: number of threads,
   K: Gaussian order.
1: for all threads i ← 0 to n/NTh do
2:    s ← 0;
3:    for all threads k ← 1 to K/NTh do
4:       r_k ← p_k × R(C_{n_i} | λ_k Σ_k);
5:       s ← s + r + k;
6:    for all threads k ← 1 to K/NTh do
7:       w_ik ← r_k / s;
8: return w_ik
```

Algorithm 9 gmmMTrain (w_{ik}, C_n, n, NTh)

```
Require:
    w_ik: density function,
    C_n : MFCC vector,
    n: vector length,
    NTh: number of threads,
    K: Gaussian order.
1: N_k ← 0, p ← 0, λ ← 0, σ ← 0;
2: for all threads i ← 1 to n/NTh do
3:     for k ← 1 to K/NTh do
4:         N_k ← N_k + r_ik;
5: for all threads k ←1 to K/NTh do
6:     p_k ← N_k/N;
7: for all threads i ← 1 to N do
8:     for k ← 1 to K/NTh do
9:         for m ← 1 to P/NTh do
10:            λ_kp ← λ_kp + r_ik × x_id/N_k;
11: for all threads i ← 1 to N do
12:    for k ← 1 to K/NTh do
13:        for m ← 1 to P/NTh do
14:            σ²_km ← σ²_km + r_ik × (x_im − λ_im)/N_k;
15: for all threads i ← 1 to K/NTh do
16:    ∑_k ← diag(σ²_k);
17: for all threads i ← 1 to K/NTh do
18:    llh ← llh+ log R(C_ni|λ_k, ∑_k);
19: return llh
```

between the computed log-likelihood values in the last iterations happens. The best-found GMM parameters will be saved to be used in the testing phase.

Algorithm 8 shows the parallel code for the E-step during the training process, where k Gaussian mixing coefficients, k Gaussian means, and k Gaussian covariances are computed using Eqs. (8)–(10), respectively, for all the data points of dimension n. In each loop, the data points are distributed evenly to threads, and each thread performs the computation in parallel.

Algorithm 9 shows the code for the M-step during the training process, where the membership of the data points is changed. k Gaussian mixing coefficients, k Gaussian means, and k Gaussian covariances are computed again for all the data points of dimension P. Using Eq. (14), the log-likelihood is computed.

5 Experimental Result

To evaluate the ASR algorithms, the training experiments are conducted using a collection of TIMIT Acoustic-Phonetic Continuous Speech Corpus; TIMIT contains broadband recordings of 630 speakers of eight significant dialects of American English, each reading ten phonetically rich sentences. The TIMIT corpus includes time-aligned orthographic, phonetic, and word transcriptions, and a 16-bit 16 kHz speech waveform file for each utterance[39]. For training, only 20 males and 16 females speakers of speech waveforms are used, with a total duration of more than 2 hours of speech; the minimum period of speech signal for each speaker is around 3 minutes. Only short sentences of a maximum time of 0.04 seconds are used for testing, not in the testing phase. In our implementation, we apply the same ASR configuration for the comparison offered:

- The speech is segmented into 20 ms windows with overlapped between frames of 10 ms.
- The speech frame is pre-emphasized using a 0.95 pre-emphasis coefficient before being windowed by a Hamming window.
- A speech activity detector is then used to discard silence-noise frames. The speech activity detector is a self-normalizing and energy-based detector that tracks the signal's noise floor to adapt it to noise conditions.
- Next, Mel-scale Cepstral feature vectors are extracted from the speech frames. The Mel-scale Cepstrum is the discrete cosine transform of the log-spectral energies of the speech segment Y, where all Cepstral coefficients except its zero values (the DC level of the log-spectral energies) are retained in the processing.
- Delta Cepstral are computed using a first-order orthogonal polynomial temporal fit over two feature vectors.
- A twelve-order GMM is used for feature aggregation[40]. A maximum of $M = 10\,000$ iterations are used to find the full log-likelihood. The trained data are stored for each speaker and used for testing prepossess.

All the experiments are performed with 1, 2, 4, and 8 threads using Intel Core i5- 6300HQ, 2.3 GHz CPU, with 8 GB RAM and without hyper-threading under Ubuntu 14.04 LTS operating system, kernel version number RPC 3.19.0-73. Different implementations of the ASR are performed:

- The sequential version uses Matlab programming language to study the overall performance of speaker recognition accuracy related to the used speech corpus accuracy[41] and time consumption during the training and testing phases.
- The sequential version uses C programming language version 4.8.4[37].
- Parallel versions of the sequential C code using OpenMP and PThreads libraries.

Figure 4 compares the performance of the ASR workload for both MATLAB and C codes during the training and testing phases. As expected, the results show the superior performance of C implementation in terms of running time in both stages.

Figure 5 shows the speed-up[23] for PThreads and OpenMP using 1, 2, 4, and 8 threads for both the training (MFCC and GMM stages, see Fig. 5a and testing phases (MFCC and GMM stages, see Fig. 5b. The best sequential running time was 856 s for training 2 hours of speech signals and 11 s for detecting 36 speakers.

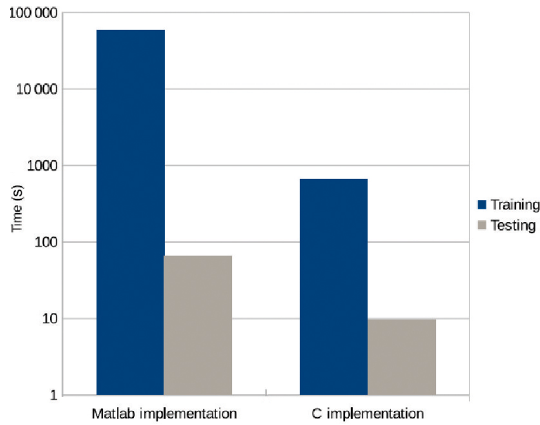
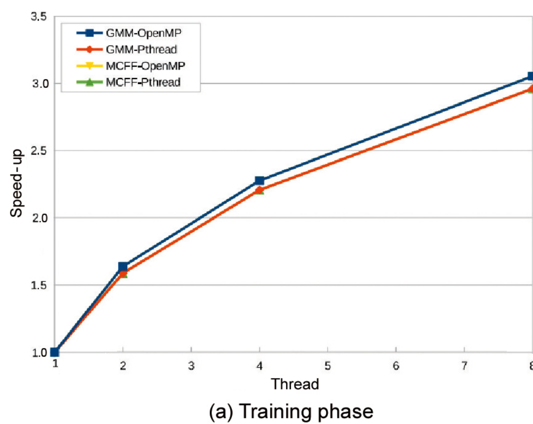


Fig. 4 Performance comparison of the C and Matlab ASR codes for the testing and training phases.



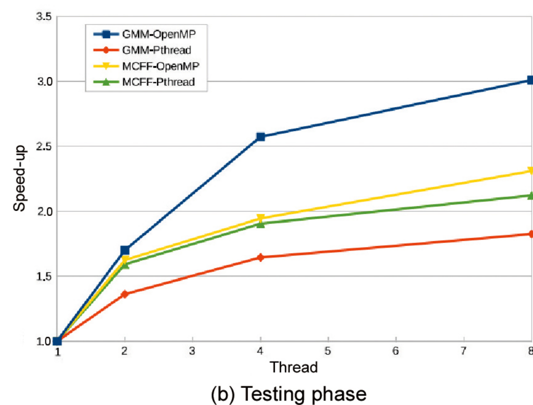


Fig. 5 Achieved speed-up using PThreads and OpenMP for a fixed instance of the problem (no differences are noticed in speed-up between GMM and MFCC in the training phase).

The results show that both parallel versions show improvements in the running time of the ASR system. The improvement is almost identical in the training stage for both PThread and OpenMP versions. The improvement is better for OpenMP than PThreads in the testing stage. This is because PThreads introduce additional overhead due to the use of locks to protect critical sections, increasing the idle time of the threads. The idle time is unnoticeable during the training phase because the amount of data to be processed is much more significant than in the testing phase[27].

Figure 6 shows the overall performance of the ASR during the testing and training stages for the OpenMP version. The effect of increasing the number of threads to more than 4, especially during the testing phase (recognition), is not noticeable. This is because of the serial part of the code, which is more important when the amount of data to be processed is relatively low. Following the Amdahl Law[23], the percentage of parallel code in ASR is approximately 75%[23].

We also measure both PThreads and OpenMP versions regarding speaker recognition accuracy during testing. For this purpose, we make a coarse test using different signal durations to study the effect of ASR recognition accuracy on buffer time. The speech duration is established to achieve good recognition accuracy. Using the tested speech corpus[40], the speech utterance duration used in the speaker recognition is 3.5, 1.7, 0.9, 0.5, and 0.15 s. The detection results are shown in Fig. 7, where even with a speech duration of around 0.5 s (relatively far from one spoken word), the detection rate is relatively high (near 98%). The accuracy is 100%

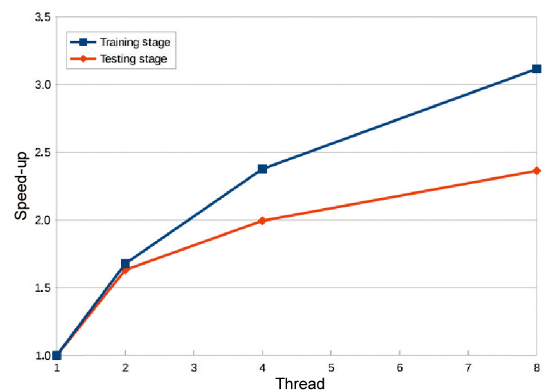


Fig. 6 Speed-up of the OpenMP version for the overall training and testing stages.

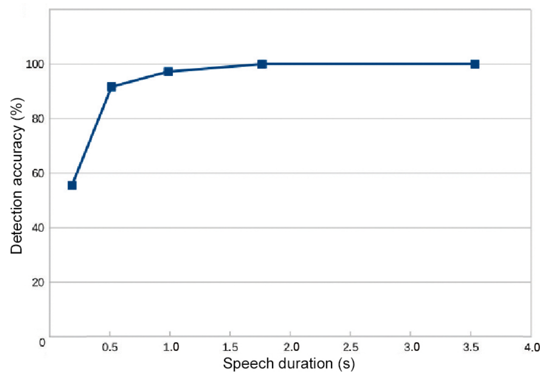


Fig. 7 Accuracy of the ASR for both PThreads and OpenMP versions with eight threads.

when the speech duration is more significant than around 1.7 s. Therefore, it is only sufficient for the tested speaker to speak one or two words to be recognized.

6 Conclusion and Future Work

This paper uses OpenMP and PThreads libraries to code parallel robust speaker recognition versions. We have used two main benchmarks with different sub-stages covering essential parts of ASR for their evaluation. The results show a better performance using OpenMP with a speed-up of around 3.2, achieved on the personal laptop with an i5 CPU (4 cores), despite the sequential dependencies in the ASR algorithm.

The remarkable performance of ASR in recognition rate, even using a concise duration of the spoken sentence of the unknown speaker, could be achieved using a high number of speech features in the parallel version. In future work, the parallel algorithms will be further tested with a large speaker population, under different speech quality, and using a more significant number of cores.

References

[1] T. Kinnunen and H. Li, An overview of text-independent speaker recognition: From features to supervectors, *Speech Commun.*, vol. 52, no. 1, pp. 12–40, 2010.

[2] D. A. Reynolds, Automatic speaker recognition using Gaussian mixture speaker models, *Lincoln Lab. J.*, vol. 8, no. 2, pp. 173–191, 1995.

[3] R. Auckenthaler, E. S. Parris, and M. J. Caray, Improving a GMM speaker verification system by phonetic weighting, in *Proc. IEEE Int. Conf. Acoustics, Speech, and Signal Processing*, Phoenix, AZ, USA, 1999, pp. 313–316.

[4] A. Janicki and S. Biay, Improving GMM-based speaker recognition using trained voice activity detection, https://www.researchgate.net/publication/268290565_Improving_GMM-based_Speaker_Recognition_Using_Trained_Voice_Activity_Detection, 2006.

[5] D. A. Reynolds, T. F. Quatieri, and R. B. Dunn, Speaker verification using adapted Gaussian mixture models, *Digital Signal Processing*, vol. 10, nos. 1–3, pp. 19–41, 2000.

[6] F. Ganjeizadeh, H. Lei, A. Maganito, and G. Pallipatta, Reducing the computational complexity of the GMM-UBM speaker recognition approach, *Int. J. Eng. Res. Technol.*, vol. 3, no. 3, pp. 1793–1797, 2014.

[7] R. Makhijani, U. Shrawankar, and V. M. Thakare, Opportunities & challenges in automatic speech recognition, arXiv preprint arXiv:1305.2846, 2013.

[8] M. Petracca, A. Servetti, and J. C. De Martin, Low-complexity automatic speaker recognition in the compressed GSM AMR domain, in *Proc. IEEE Int. Conf. Multimedia and Expo*, Amsterdam, the Netherlands, 2005, p. 4.

[9] E. Gonina, G. Friedland, H. Cook, and K. Keutzer, Fast speaker diarization using a high-level scripting language, in *Proc. IEEE Workshop on Automatic Speech Recognition and Understanding*, Waikoloa, HI, USA, 2011, pp. 553–558.

[10] D. A. Reynolds and R. C. Rose, Robust text-independent speaker identification using Gaussian mixture speaker models, *IEEE Trans. Speech Audio Process.*, vol. 3, no. 1, pp. 72–83, 1995.

[11] T. Yoshimura, T. Fujimoto, K. Oura, and K. Tokuda, SPTK4: An open-source software toolkit for speech signal processing, presented at Proc. 12th Speech Synthesis Workshop, Grenoble, France, 2023.

[12] P. Pacheco, *An Introduction to Parallel Programming*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[13] IEEE Standard for Information Technology: Portable Operating System Interface (POSIX), https://pubs.opengroup.org/onlinepubs/009695399/, 2022.

[14] F. Bimbot, J. F. Bonastre, C. Fredouille, G. Gravier, I. Magrin-Chagnolleau, S. Meignier, T. Merlin, J. Ortega-García, D. Petrovska-Delacrétaz, and D. A. Reynolds, A tutorial on text-independent speaker verification, *EURASIP J. Adv. Signal Process.*, vol. 2004, pp. 430–451, 2004.

[15] R. N. Bracewell, *The Fourier Transform and Its Applications*. New York, NY, USA: McGraw-Hill, 1965.

[16] L. R. Rabiner, A tutorial on hidden Markov models and selected applications in speech recognition, in *Proc. IEEE*, vol. 77, no. 2, pp. 257–286, 1989.

[17] J. Vanek, J. Trmal, J. V. Psutka, and J. Psutka, Optimization of the Gaussian mixture model evaluation on GPU, in *Proc. Interspeech 2011*, Florence, Italy, 2011, pp. 1737–1740.

[18] G. Friedland, J. Chong, and A. Janin, Parallelizing speaker-attributed speech recognition for meeting browsing, in *Proc. IEEE Int. Symp. Multimedia*, Taichung, China, 2010, pp. 121–128.

[19] A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing*. Englewood Cliffs, NJ, USA: Prentice-Hall, 1989.

[20] W. J. J. Roberts and J. P. Willmore, Automatic speaker recognition using Gaussian mixture models, in *Proc. Information, Decision and Control Data and Information Fusion Symp., Signal Processing and Communications Symp. and Decision and Control Symp.*, Adelaide, Australia, 1999, pp. 465–470.

[21] D. Reynolds, Gaussian mixture models, in *Encyclopedia of Biometrics*, S. Z. Li and A. Jain, eds. New York, NY, USA: Springer, 2009, pp. 659–663.

[22] F. Pernkopf and D. Bouchaffra, Genetic-based EM algorithm for learning Gaussian mixture models, *IEEE Trans. Pattern Anal. Machine Intell.*, vol. 27, no. 8, pp. 1344–1348, 2005.

[23] G. M. Amdahl, Validity of the single processor approach to achieving large scale computing capabilities, *AFIPS Conf. Proc.*, vol. 30, pp. 483–485, 1967.

[24] C. M. Bishop, *Pattern Recognition and Machine Learning*. New York, NY, USA: Springer, 2006.

[25] C. E. Leiserson and I. B. Mirman, *How to Survive the Multicore Software Revolution* (*or at Least Survive the Hype*). Burlington, MA, USA: CILK Arts, Inc., 2008.

[26] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and J. Shun, Internally deterministic parallel algorithms can be fast, in *Proc. 17th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, New Orleans, LA, USA, 2012, pp. 181–192.

[27] L. Dagum and R. Menon, OpenMP: An industry standard API for shared-memory programming, *IEEE Comput. Sci. Eng.*, vol. 5, no. 1, pp. 46–55, 1998.

[28] C. Pheatt, Intel® threading building blocks, *J. Comput. Sci. Coll.*, vol. 23, no. 4, p. 298, 2008.

[29] Z. Budimlić, V. Cavé, R. Raman, J. Shirako, S. Taşırlar, J. Zhao, and V. Sarkar, The design and implementation of the habanero-java parallel programming language, in *Proc. ACM Int. Conf. Companion on Object Oriented Programming Systems Languages and Applications Companion*, Portland, OR, USA, 2011, pp. 185&186.

[30] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, X10: An object-oriented approach to non-uniform cluster computing, in *Proc. 20th Annu. ACM SIGPLAN Conf. Object-Oriented* it Programming, Systems, Languages, and Applications, San Diego, CA, USA, 2005, pp. 519–538.

[31] R. D. Blumofe and C. E. Leiserson, Scheduling multithreaded computations by work stealing, *J. ACM*, vol. 46, no. 5, pp. 720–748, 1999.

[32] R. J. Anderson and L. Snyder, A comparison of shared and nonshared memory models of parallel computation, in *Proc. IEEE*, vol. 79, no. 4, pp. 480–487, 1991.

[33] M. Andersch, C. C. Chi, and B. Juurlink, Using OpenMP superscalar for parallelization of embedded and consumer applications, in *Proc. Int. Conf. Embedded Computer Systems*, Samos, Greece, 2012, pp. 23–32.

[34] J. Arndt, Algorithms for programmers ideas and source code, http://www.jjj.de/fxt/, 2015.

[35] M. J. Quinn, *Parallel Programming in C with MPI and OpenMP*. Boston, MA, USA: McGraw-Hill Higher Education, 2004.

[36] OpenMP: Application Programming. Interface. Version 4.5 November 2015, https://pubs.opengroup.org/onlinepubs/009695399/, 2022.

[37] W. P. Petersen and P. Arbenz, *Introduction to Parallel Computing: A Practical Guide with Examples in C*. Oxford, UK: Oxford University Press, 2004.

[38] K. R. Rao and P. Yip, *Discrete Cosine Transform: Algorithms, Advantages, Applications*. San Diego, CA, USA: Academic Press, 1990.

[39] N. S. Disc, J. S. Garofolo, L. F. Lamel, W. M. Fisher, J. G. Fiscus, D. S. Pallett, and N. L. Dahlgren, Acoustic-phonetic continuous speech corpus, https://catalog.ldc.upenn.edu/LDC93s1, 2022.

[40] J. Vaněk, J. Trmal, J. V. Psutka, and J. Psutka, Full covariance Gaussian mixture models evaluation on GPU, in *Proc. IEEE Int. Symp. Signal Processing and Information Technology*, Ho Chi Minh City, Vietnam, 2012, pp. 203–207.

[41] L. Lu, A. Ghoshal, and S. Renals, Acoustic data-driven pronunciation lexicon for large vocabulary speech recognition, in *Proc. IEEE Workshop on Automatic Speech Recognition and Understanding*, Olomouc, Czech Republic, 2013, pp. 374–379.

Talha Ali Khan is a professor of data science and program director of data science at University of Europe for Applied Sciences, Postdam, Germany. He received the BEng degree in electronics engineering from NED University of Engineering & Technology, Pakistan in 2009, the MEng degree from King Saud University, Riyadh, Saudi Arabia, and the PhD degree from the University of Technology Sydney, Australia in 2012. His research interests are in optimisation, algorithms, artificial intelligence, and data minning.

Rand Kouatly is a professor of information technology & communication and program director of software engineering at University of Europe for Applied Sciences, Postdam, Germany. He received the BEng degree in telecommunication engineering from Damascus University, Syria in 1991, and the MEng and PhD degrees from Ain Shams University, Egypt in 1994 and 2000, respectively. His main research areas are speech and speaker recognition, artificial intelligence, and data analytics.