7-27-2001

A master-slave architecture for parallel speaker recognition

Sunil Kumar Godavarthi
Florida International University

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

A MASTER-SLAVE ARCHITECTURE FOR PARALLEL SPEAKER RECOGNITION

A thesis submitted in partial fulfillment of the

requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Sunil Kumar Godavarthi

2001

To: Dean Arthur W. Herriott
College of Arts and Sciences

This thesis, written by Sunil Kumar Godavarthi, and entitled A Master-Slave Architecture for Parallel Speaker Recognition, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this thesis and recommend that it be approved.

Shu-Ching Chen

Masoud Milani

Marie Roch, Major Professor

Date of Defense: July 27, 2001

The thesis of Sunil Kumar Godavarthi is approved.

Dean Arthur W. Herriott
College of Arts and Sciences

Dean Douglas Wartzok
Graduate School

Florida International University, 2001

ACKNOWLEDGMENTS

I would like to thank Dr. Marie Roch, my advisor, for having the faith and confidence in me to do this project and for her constant guidance and support throughout the thesis. I also thank her for reading through my thesis for all those corrections. This thesis could not have been completed without her help. I would also like to express my gratitude and appreciation to the following people:

To my thesis committee members, Dr. Chen and Dr. Milani. To my colleagues and friends at CADSE and HPDRC Manish Mahajan, Suresh Cheggi Reddy, and George McGivan for their constant support, motivation and encouragement. To my roommates and friends here in Miami for being my personal cheerleading squad. To my parents and brother for their constant support and their trust in me.

ABSTRACT OF THE THESIS

A MASTER-SLAVE ARCHITECTURE FOR PARALLEL SPEAKER RECOGNITION

by

Sunil Kumar Godavarthi

Florida International University, 2001

Miami, Florida

Professor Marie Roch, Major Professor

Speaker recognition is one of the popular research interests in speech processing. A speaker recognition system receives the speech signal (data) and determines who the speaker is from a known set of speakers. This process involves the task of matching the input speech signal to the models for all the speakers enrolled in the system. Important factors that determine the success of these systems are response time and accuracy.

The objective of my thesis is to optimize response time by dividing the task of recognition into a number of sub tasks and to execute these individual tasks on load-balanced multiple processors. There has been limited research in improving the response time with the use of parallelism. This idea has been implemented by using a master-slave model in which the master divides the recognition tasks and initiates their parallel processing on multiple slaves. Tests performed showed that the response time achieved is better than those obtained from the conventional system, which does not involve any parallel processing. This thesis justifies that a parallel processing approach can be used to optimize the response time of a speaker recognition system.

TABLE OF CONTENTS

LIST OF TABLES

TABLE PAGE

LIST OF FIGURES

FIGURE PAGE

1. Introduction

A popular research interest for speech signals is speech and speaker recognition. Speech recognition is the process of deciphering the speech signal by a machine to recognize the sequence of words a person is saying, whereas speaker recognition is the process of identifying the speaker by a machine. Some topics involve both these technologies (recognizing which speaker is saying what in a group)[1].

A typical speaker recognition system consists of a set of N models trained for a set of N speakers (one model per speaker). This kind of system permits efficient adding, deleting and adapting of speakers to the system as opposed to a single monolithic system which requires the data used to train the previous models and large computation time. The recognition can be in set or out of set. In-set recognition refers to the process in which the speakers to be recognized are from the set of models that were used to train the system. The out-of-set recognition refers to the process in which the speakers to be recognized may or may not be from the set of models that was used to train the system.

Speaker recognition can be text dependent or text independent. In text-dependent speaker recognition the identification can only be done for a specific phrase to which the system was trained whereas text-independent recognition is independent of the training speech. The speaker recognition system receives the speech signal (data) and determines who the speaker is from a set of speakers. This process involves the task extracting relevant parts of the speech to obtain the feature data and matching this data to all the trained models in the system. One of the most popular feature extraction techniques is the

cepstrum [12]. A series of complex operations such as framing, windowing functions, fourier and logarithmic transformations are involved in the cepstrum process. The matching process would be a computation performed on the feature data and the model. This computation depends on model characterization which can be in many forms such as template matching, hidden Markov models (HMMs) or vector quantization. Thus, the time taken by the system increases with the number of speakers. This increase is linear in the case of single classification but in the case of classifying an entire group it becomes exponential. The increase in the computation time means an increase in the response time of the system. This increase may not be desirable in most commercial applications.

1.1 Purpose

Many practical speaker recognition applications have large number of speaker sets (i.e. large model sets). The most important factors that determine the success of these systems would be response time and accuracy. Optimal response time makes speaker recognition system used in most practical identification and verification applications. Response time has been of major interest for many researchers in speaker recognition. This has triggered a quest for speaker recognition techniques whose response time is less dependent on the number of models in the system. An intuitive way of increasing the computation speed is to use multiple processors, which gives birth to the idea of implementing the system on parallel processors.

1.2 Previous Work

There has been little work using parallelism in this area and most of the work that has been done, is in the field of speech recognition. Some of the relevant works in speech recognition are:

Mitchell et al. [3] proposed a method to model state duration in HMMs. Due to the increased complexity associated with this model, they used a parallel implementation to decrease the computation time that would make the testing of large systems in optimal time feasible. Kwong et al. [2] proposed a parallel genetic algorithm for training. This genetic algorithm achieved global maxima for HMM model parameters in training. Kobayashi et al. [4] proposed a method by combining two techniques. They used spectral subtraction for initial hypothesis and parallel HMM for probability calculations. Noda et al. [5] proposed a parallel technique using Markov random fields (MRFs). This method makes use of local parallel operations to be performed on speech frames over time to estimate the optimal state sequence, which speed up the recognition process.

1.3 Objective

An alternative and similar kind of solution for a speaker recognition system would be to perform the process in parallel on multiple processors with load-balancing. There is currently less interest in parallel implementations of speech or speaker recognition. The most likely reason for this is due to the fact that good pruning techniques can yield better performance than the parallel computation technique [7]. Pruning techniques decrease the computational load by reducing the number of models on which the computation needs to

be performed. Thus, less interest was generated on alternative techniques such as parallel processing. In large-scale applications where a large number of speakers are being classified, the situation may require more than good pruning techniques alone to reduce computational time. This drives the need for exploring additional techniques such as parallel processing, which would permit the use of more sophisticated, accurate and slower recognition models to be used. The purpose of this thesis is to implement a working model of a parallel speaker recognition system, which would be a solution for large-scale speaker recognition systems. This study would also reveal the factor by which the response time would be enhanced by parallelism.

The objective of this thesis can be summarized as follows:

1. To study the characteristics of the speaker recognition process in order to be able to apply parallel algorithms.
2. To study different load-balancing algorithms and implement the most suitable to the parallel speaker recognition problem.
3. To parallelize the speaker recognition process to reduce throughput time.
4. To study the factors that contributed to the improvement of throughput time.

1.4 Research Methodology

A working version of a serial text-independent speaker recognition system is in existence as a result of Dr. Roch's research work. This system implements vector quantization, HMM, and integral decode HMM recognition [12]. The majority of this

system is developed in Matlab with bottleneck sections coded in portable C. The idea of my thesis is to make this speaker recognition system to run in parallel on multiple processors. This extension of the system is primarily developed in Java and C, as Matlab has interfaces to both these languages. The reason Java was selected to implement the parallel speaker recognizer is due to the fact that Java provides a modular, platform independent solution with TCP/IP sockets for communicating between the parallel processes.

A semi-distributed approach has been adopted for implementing the parallel recognizer. This parallel recognizer has a central processor called the controller that distributes and balances load among the node processors. The node processors perform the computations and return the results to the controller. The controller frees the node processes when the processor on which it is executing is overloaded and moves the load associated with them to other node processors that are not overloaded. The average number of jobs in the run queue over a five minute period are taken to determine the load of the processor.

1.5 Significance of the thesis

Although speaker recognition is a well-studied problem, there has not been much work done on solving the long response time associated with large systems. This thesis provides algorithms, which would help the on-going research on reducing the execution time associated with large speaker sets. Thus this is an implementation of a low-cost, high-efficient mechanism for quick response of speaker recognition problems.

The interface between the different languages used in this parallel speaker recognition system provides the ideas for interfacing with new technologies and also makes the system portable and flexible. The idea of implementing parts of the system with different technologies and interfacing between them will be studied in detail. One other issue that is given significant thought will be the passing of model data between different technologies. Since Java is being used for implementing the parallel and load-balancing part, this section of the system would be architecture, operating system and platform independent. Communication and data passing between various languages will also be studied in detail.

1.6 Feature Extraction

The input data to a speaker recognition system is a pulse-code modulated speech signal. In its raw form this speech data is of little help in the process of pattern recognition, thus we exploit this input signal to get the important parts (features) which will help the process of pattern recognition. Many types of feature extraction techniques are in existence; one of the most popular and effective one is the cesptrum. Derivation of the cesptrum is a multi-step procedure. The process starts with endpointing the input speech signal. This is done to eliminate silence, sporadic external sounds and other sounds from the mouth of the speaker such as non speech related sounds produced by the movement of lips. This process is usually done using a finite state machine with transitions based on signal energy. The finite state machine changes to a high-energy state when the energy of the input signal is above a threshold and remains in that state as

long as the signal energy is above the threshold. All the signal parts that make the finite state machine stay in the high energy states for more than a certain time interval are taken as speech signal and the rest of the signal part is assumed not a part of the speaker speech.

After endpointing, the signal is broken into little byte size chunks called frames; typically these chunks overlap. Then a windowing function is applied to each of the frames. There are a number of windowing functions with different properties, some of the most popular ones are the hanning and hamming. The cepstrum uses the hamming window function, which is basically a point-by-point multiplication applied to the signal. Figure 1 below is an example of a 128 point hamming window. The hamming window process shrinks the signal at the beginning and end of each frame. This avoids large discontinuities. This elimination of large discontinuities is done because Fourier transforms on these frames assume that these frame signals repeats infinitely and large discontinuities are interpreted as huge amount of energy, but this energy in reality does not exist.

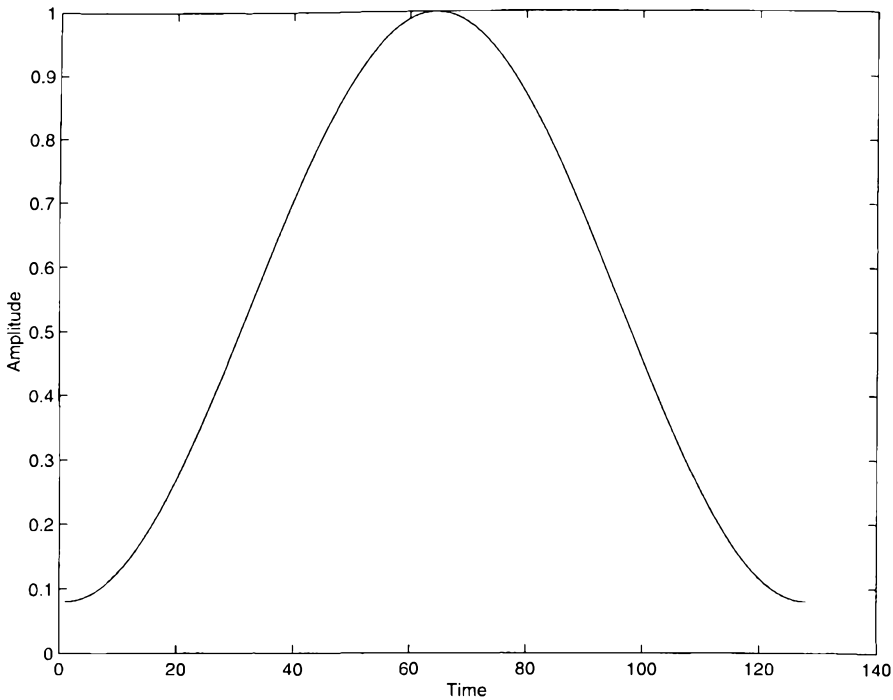

Figure 1. Hamming Window Function

The next step involves applying a Fourier transform to each of the frames. Now in the frequency domain, we multiply the signal with its complex conjugate to get the squared magnitude signal, and at this point we lose the phase information. This loss is acceptable because most speech scientists are interested in how things are heard and perceived, and it is believed that the phase information in not important. The last steps are to take the logarithm of the squared magnitude signal and to apply an inverse Fourier transform which brings the signal back to the time domain, but it is not the same exact signal as logarithms were taken. The reason this is effective is that it makes the separation of the two sounds from two sources, the signal from the lungs and vocal folds and the

vocal tract. The pressure from the lungs passing through the vocal folds of the speaker creates the impulse stream and the vocal tract is the other, which comprises of the sounds produced by the nasal and mouth cavities and articulators such as lips and tongue. In the feature extraction process we are more interested in the vocal tract, thus this process makes the separation of the source and the vocal tract easier.

1.7 Speech Database – King Corpus

To test the recognition system there is a need of a speech database consisting of speech samples from a set of speakers. Such a database used to model the speaker's speech is usually termed as corpus. A corpus known as the king95 corpus [10] was used to train and test the recognition system. The king corpus was created from all male speakers at two different locations for research purposes. The speech was recorded in ten different sessions. Each session being taken place in different time intervals and the topics to be spoken were changed from session to session.

All the sessions were recorded on two channels, namely wide-band and narrow-band. In each session the speaker is locked inside a quiet (not a sound proof) room with a headset to listen to the questions asked by other person located outside the room. The speaker hears the questions through the headset and answers. The response of the speaker is then recorded using two different microphones, a high quality and a normal telephone microphone. The speech recorded through the high-quality microphone corresponds to a wide-band channel. The narrow-band channel corresponds to the speech that has been received by a normal telephone and passed through the telephone exchanges and lines

just as a normal long distance telephone call. In our experiments this narrow band portion of the corpus is being used.

1.8 Speaker Modeling – Hidden Markov Model (HMM)

The modeling of the speakers is done using Hidden Markov Models (HMM)[6].

A hidden Markov model is created for each speaker in the system. A HMM is a probabilistic finite state machine. HMMs are extensions of Markov models. A Markov model is a set of state transition probabilities for each instance of time, where states are observable events. This Markov model is extended to the hidden Markov model by taking the observable events to be probabilistic function of the state rather than as an observable event. It is called hidden because each state of the HMM is hidden, and is only visible by an other probabilistic function as the states in the HMM's are probabilistic functions. Formally, HMM is defined by Rabiner [6] as a two-fold stochastic process in which one stochastic process is hidden and is observable only by an other stochastic process. There are three basic problems associated with the HMM's:

- Given an observation sequence O and a model λ how do we efficiently compute the probability of the observation sequence.
- Given an observation sequence O and a model λ how do we choose the optimal state sequence q.
- How to maximize an observation sequence by adjusting the model parameters.

just as a normal long distance telephone call. In our experiments this narrow band portion of the corpus is being used.

1.8 Speaker Modeling – Hidden Markov Model (HMM)

The modeling of the speakers is done using Hidden Markov Models (HMM)[6].

A hidden Markov model is created for each speaker in the system. A HMM is a probabilistic finite state machine. HMMs are extensions of Markov models. A Markov model is a set of state transition probabilities for each instance of time, where states are observable events. This Markov model is extended to the hidden Markov model by taking the observable events to be probabilistic function of the state rather than as an observable event. It is called hidden because each state of the HMM is hidden, and is only visible by an other probabilistic function as the states in the HMM's are probabilistic functions. Formally, HMM is defined by Rabiner [6] as a two-fold stochastic process in which one stochastic process is hidden and is observable only by an other stochastic process. There are three basic problems associated with the HMM's:

- Given an observation sequence O and a model λ how do we efficiently compute the probability of the observation sequence.
- Given an observation sequence O and a model λ how do we choose the optimal state sequence q.
- How to maximize an observation sequence by adjusting the model parameters.

The solutions to these problems are explained in detail by Rabiner [6]. Below is the outline of these solutions. The same naming conventions have been followed for clarity.

The first problem of computing the probability of the observation sequence $O = (o_1\ o_2\ o_3 \ldots\ldots o_T)$, has recursive methods called the forward and backward procedures. The forward procedure follows on the concept of calculating the probabilities of all paths into a particular state at time t summed into a unit and this is taken as the probability of reaching that state. This sum of probability is again used to calculate the probability of reaching the succeeding states at time t+1. This is done upto a time T, and the probabilities of all states are summed at time T. The sum of all these probabilities gives the probability of the observation sequence over the model. The backward procedure is the same as the forward, expect for the fact that the probabilities of reaching different state at time T-1 is calculated first and the process goes in reverse direction.

The solution to the second problem is the Viterbi algorithm. The is the algorithm used during parallel computing. This algorithm gives the best state sequence $q = (q_1\ q_2\ q_3 \ldots\ldots\ q_T)$ for a given observation sequence $O = (o_1\ o_2\ o_3 \ldots\ldots o_T)$. In order to find this best state sequence we define $\delta_t(i)$ such that after t observations in the given observation sequence, the highest probability is to reach the state i. To keep these highest probability states and form the best state sequence we need to maintain another array denoted by $\Psi_t(j)$ for all t ($2 <= t <= T$), such that it gives the probability of the best path into state j at time t.

Thus the $\delta_t(j)$ and $\Psi_t(j)$ are given by the expressions:

$$\delta_t(j) = \max_{1<=i<=N} [\delta_{t-1}(i)\ a_{ij}]\ b_j(o_t),$$

$\Psi_t(j) = \arg\max_{1<=i<=N} [\delta_{t-1}(i)\, a_{ij}]$, where $2<= t <= T$ and $1<=j<=N$

a_{ij} – is the probability of moving from the ith state to the jth state and $b_j(o_t)$ – is the probability of being in state j at the observation o_t in the observation sequence. All the Ψ_t for t = T-1, T-2, ..., 1. give the optimal observation sequence.

The third problem of adjusting the model parameters is related to the process of training the models for the recognition system. The expectation-maximization method is used as a solution for this problem. First a good initial guess is made on the model parameters. Next the model is adapted to the to best suit the training data. This adaptation is done by adjusting the model parameters around the mean of the model data that is being used for training. The above process iterates for the number of mixtures specified. This algorithm guaranties convergence to a load maximum.

2. Architecture

2.1 Overview

The proposed architecture for a parallel speaker recognizer is based on a master-slave mechanism. The basic concept is that the master distributes the recognition tasks to the slaves and the slaves perform the tasks and return the result to the master when the computation is completed. The goal is to decrease recognition time by breaking up the task into smaller units and executing them in parallel on different processors.

2.2 Interface to the Existing System

The existing system is a serial implementation of the speaker recognition system. This system thus runs on a single processor that determines which feature data needs to be tested against the models and calls C code to perform the scoring of each model. The C code is used to compute the likelihood of observing the feature data with respect to the models. The speech utterances which make up the feature data, and the model data which corresponds to the speakers are sent to the C code for the HMM likelihood calculations. This process of likelihood calculation needs to be performed for each of the segments of an utterance with respect to the models determined by the system. As a result, a large number of computations are generated on a single processor. Thus a parallel system is designed to execute these computations on multiple processors in parallel.

The new parallel system interfaces with the existing one at the point where the testing process for the speakers begins. The master is hosted on the processor as a Matlab processor and the model and feature data are registered with the master. Upon creation, the master invokes slaves on different processors by using remote shell. Then the Matlab process determines for each speaker which utterances (feature data) needs to be tested. For each of these test utterances, the Matlab process determines which segments (tokens) of the utterance needs to be tested with what models. Every test token and model pair requires a call to the C function. The Matlab process encapsulates these tasks by registering jobs on the master. Each job consists of a token and a set of models that need to be tested with respect to the token. Thus at this point there exist a number of jobs to be performed per speaker, as the likelihood calculation of the feature data need to be performed over a set of models. Now the master distributes these jobs to the slave processes and sends the required data to the slaves for job processing. The slave processes perform the jobs and return the likelihood calculations to the master, which in turn returns them to the Matlab process.

2.3 Parallel Architecture

Figure 2 below shows the parallel architecture for the speaker recognition that is based on a master-slave model. The architecture assumes one master and a group of co-operating slave running on different processors. The master runs on the same processor as the Matlab process. The slaves run on different processors and connect to the master. The master creates a connection handler for each of the connected slaves. Once the master has

connected slaves it distributes the jobs to the slave. The master and the slaves use a message object to communicate. The slaves perform the jobs by calling a native C method and return the results of each job as they are processed to the master. The master collects these results and makes them available to Matlab only when the results of all the distributed jobs are available. Then a new set of jobs for the next utterance are created by Matlab and passed on to the master for distribution and the cycle continues till all the utterances to be tested are completed.

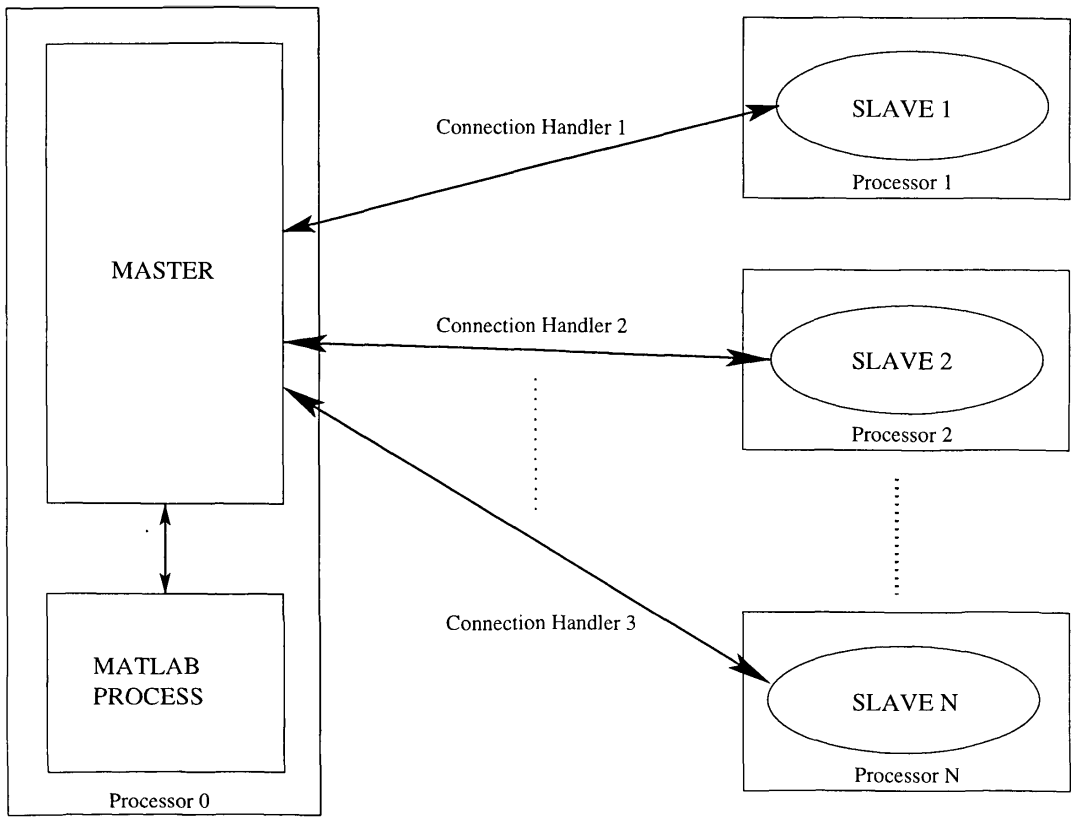

Figure 2. Master-slave Architecture

2.4 Master Architecture

The master acts as a controller, which initiates the parallel process by starting the slave processes. The primary tasks of the master are load balancing and load distribution among the slaves besides gathering results from the slaves. A separate thread of communication called the connection handler is created on the master for each initialized slave. TCP sockets are used by the connection handler for communication between the slaves and the master to ensure a reliable means of communication.

Once the slaves register themselves with the master along with the current load of their processors on with they are executing, the master does the initial load distribution. Depending on the load distribution algorithm used the master may or may not take the loads of the slaves processes into consideration for load distribution. The various load distribution algorithms are discussed below in the algorithms section. The master distributes the data of all the models on to the connected slaves and then distributes the jobs. After the distribution of the jobs the master listens to the slaves through the connection handlers by receiving message objects. The message objects encapsulate all the necessary information.

The master may receive one of the following messages: an overloaded message, a feature data request message, a deleted load message or a result message. When the master receives an overloaded message the master calls the load-balancing algorithm to adjust the load among processors. The various load-balancing algorithms that have been used to test the system are described in detail in the algorithms section. The basic concept of these load-balancing algorithms is to check if there exists at least one slave process,

which is below the overload threshold. If an under loaded slave exists, then some or all jobs (depending on the algorithm) from the overloaded slave are removed and added to the under loaded one. When a feature data request is received the master identifies which feature data is requested with the help of the feature identifier in the incoming message, and sends a message back to the slave with the feature data. When a deleted load messages is received the master saves the deleted job identifier in its own deleted jobs queue for distributing it to under-loaded slaves. The last type of message the master can receive is the result message, when a result message arrives the master saves the result in one data structure.

The types of messages that the master sends out to slave are; delete load message and add load message. Both these messages are a result of the load balancing process. When a load-balancing algorithm decides to remove the load from the overloaded slave the master sends a delete load message with the number of jobs to be deleted. The identities of the specific jobs to be deleted are left to the discretion of the slave. The add load message is sent when the master decides to add the deleted jobs of an overloaded slave to an under-loaded slave.

The master makes the results available to the Matlab process once all the distributed job results are computed by the slaves. Once the results are available the Matlab process accesses them and assigns new jobs to the master (if more utterances exit). This cycle continues till the Matlab process explicitly shuts down the master when the testing process on the utterances is complete.

The master from time to time poles to accept connections from slaves, which could not connect previously. When a new slave connects to the master at a later time, the master assigns the deleted jobs to this slave if any. Otherwise a fraction of jobs from the slave that has the highest load at that instance are removed and assigned to the new slave. This fraction depends on the load of the highly loaded slave.

2.5 Slave Architecture

The primary task of the slave is to perform the jobs assigned to it by the master. The slave invokes a separate thread for monitoring load on the processor on which it runs. This load information is sent to the master if the slave successfully connects to the master. The initial load distribution process of the master may use this load information to distribute the existing jobs by populating the job queues of the slaves.

The slave process executes the jobs from the queue on a first-in first-out manner. Before dispatching a new job the slave checks the load on the processor on which it is running. If the load is above a threshold value it enqueues the job and sends an overloaded message to the master and waits for the master's reply. Depending on the master's reply the slave removes some or all jobs or continues processing without removing any jobs. If the master sends a delete jobs message the slave dequeues the specified number of jobs form the beginning of its job queue and sends the job identifiers of these deleted jobs to the master to inform it about which jobs have been deleted. If the slave processor is not overloaded the slave dequeues the job from its job queue and checks if the required feature data are locally cached or not. If the required feature data is

not cached on the slave, a feature data request message is send to the master and the job is enqueued. When the feature is locally cached the slave processes the job and sends the result to the master. The slave may also receive an add load message from the master. When this message is received the slave enqueues the new jobs on to its job queue. When all the jobs assigned to the slave are completed it waits for the master to send new jobs.

2.6 Protocols

All the communication between the master and the slave is through the connection handlers, which run as separate threads on the master's processor. The communication is strictly through the exchange of message objects over the connection handlers. The list of different kind of messages are shown in Table 1. The slave communicates with the native function using JNI (Java Native Interface).

The slaves invoke a native method in C (that resides on the same processor) for the hidden Markov model likelihood calculations for the respective models assigned if the feature data is available. The result of the likelihood calculation of each model over a particular feature data is then sent to the master in an appropriate message format. The slave periodically reads a load variable that is updated by the monitor load thread that determines the load on the processor on which the slave is running. If the load exceeds a threshold value, a message is sent to the master indicating the overloaded state. Upon the arrival of overload messages from one or more slaves, the master performs load balancing by removing some jobs from the overloaded slaves and reassigning them to the ones that are not overloaded. For this process of load-balancing the master sends a delete

jobs message to the overloaded slave to remove a specified number of jobs, the slave return the job identifiers of the deleted jobs. These deleted job identifiers are then stored on the master, which reassigns to the slaves that are not overloaded depending of their load information

Index	Message Type	Description
1.	Message	Basic message type. Used to by the master to add jobs and to request the deletion of jobs to slaves. The slaves use it for reporting overload state and sending deleted jobs.
2.	Feature Request Message	Used by the slave to request feature data from the master.
3.	Feature Data Message	Used by the master to send requested feature data to the slave.
4.	Result Message	Used by the slave to return the model likelihood scores.

Table 1. Types of Messages

2.6.1 Job Processing

Each job in the job queue assigned to the slave consists of a feature data and a set of models with which the feature data needs to be scored. Each model is to be scored individually with the feature data which needs a native function call. Therefore each job

consists of sub-jobs that need to be processed individually. A sub-job has one feature identifier and one or more of model identifiers. The slave uses the model and feature identifiers as keys to obtain the model and feature data respectively from the slave repositories. Building function calls for each of them with the appropriate model and feature data processes the sub-jobs. The result of each function call of a sub-job are packed and sent to the master as result message for a job.

2.6.2 Slave Master Communication protocol

Figure 3 below shows the basic communication protocol between the master and the slaves. Each slave connects to the master's socket and registers itself by passing its machine name and the load of the processor on which it is executing. The master creates a connection handler for the connected slave and assigns a unique identifier to the slave. The master now sends the model data to the slave over the socket. The jobs registered on the master are distributed to according to the load distribution algorithm. A job queue consisting of the assigned jobs to the slave is built and sent over the socket. Each job in the queue is a collection of sub-jobs. The detailed description of the job processing is described in section 2.7.2. The slave dequeues a job from the job queue, identifies the sub-jobs and processes them by passing the appropriate data to the native C function using JNI. The slave checks whether a copy of the required feature data to process the job is locally cached or not. If it is not cached, then a request is sent to the master in the form of a feature data request for the desired feature data by passing the feature identifier. When the master receives a feature data request message it uses the feature identifier

from the message as a key to retrieve the feature data from its feature data repository. This feature is sent to the requesting slave. When the feature data arrives, the slave stores it in its repository with feature identifier as its key. Once all the sub-jobs within a job are complete the result of all the sub-jobs are packed sent to the master as a single result message.

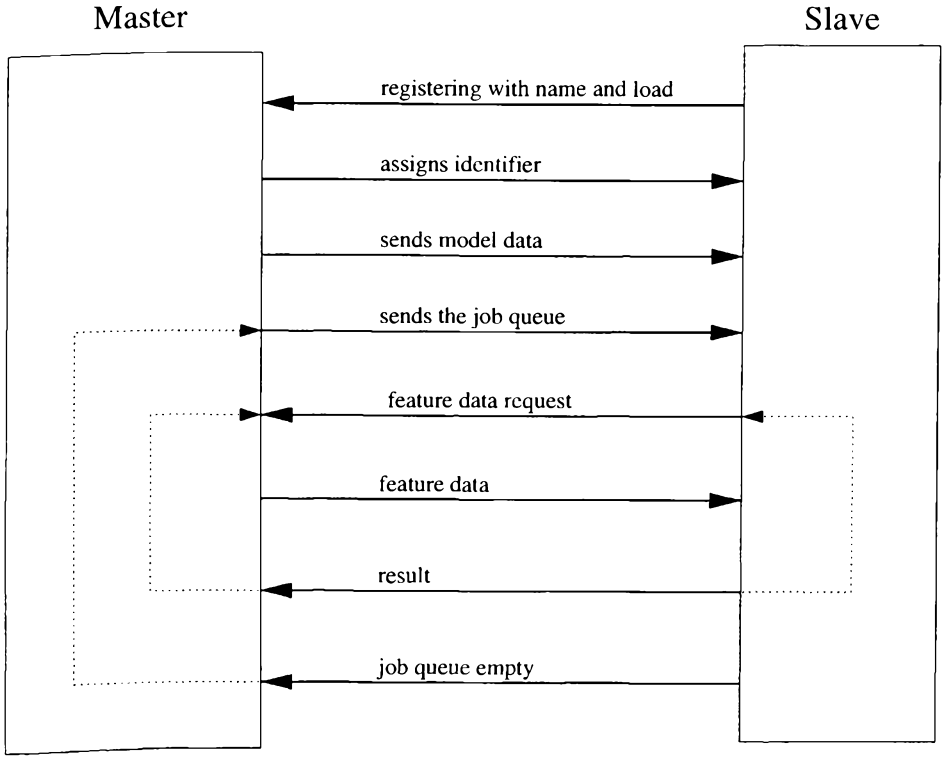

Figure 3. Slave – Master Communication Protocol

2.6.3 Slave Processor Overloaded

Figure 4 below shows the communication protocol when the slave enters an overloaded state. Every slave has a load threshold level associated with it. When a slave exceeds this threshold it sends an overloaded message to the master. The master upon receiving such a request, checks for a lightly loaded slave exists and the number of slaves which are already in overload state. If such a lightly loaded slave exists and the number of slaves in overloaded state are less than half the number of slaves connected, then the master sends a delete load message with the some or all jobs to be removed from the overloaded slave. This number is directly proportional to the current load of the overloaded slave. If a lightly loaded slave does not exist in the system a delete load message is send to the slave with zero number of jobs to be deleted.

Upon receiving a delete load message, the slave checks the number of jobs to be deleted in the message. If this number is zero, then the slave continues processing jobs. If the number is greater than zero, the slave dequeues the requested number of jobs to be deleted from its job queue and sends the identifiers of the dequeued jobs to the master in the form of deleted jobs message. Upon receiving a deleted load message, the master reads the job identifiers that have been deleted and stores the identifiers in its deleted job identifier data structure for distributing them to the lightly loaded slaves.

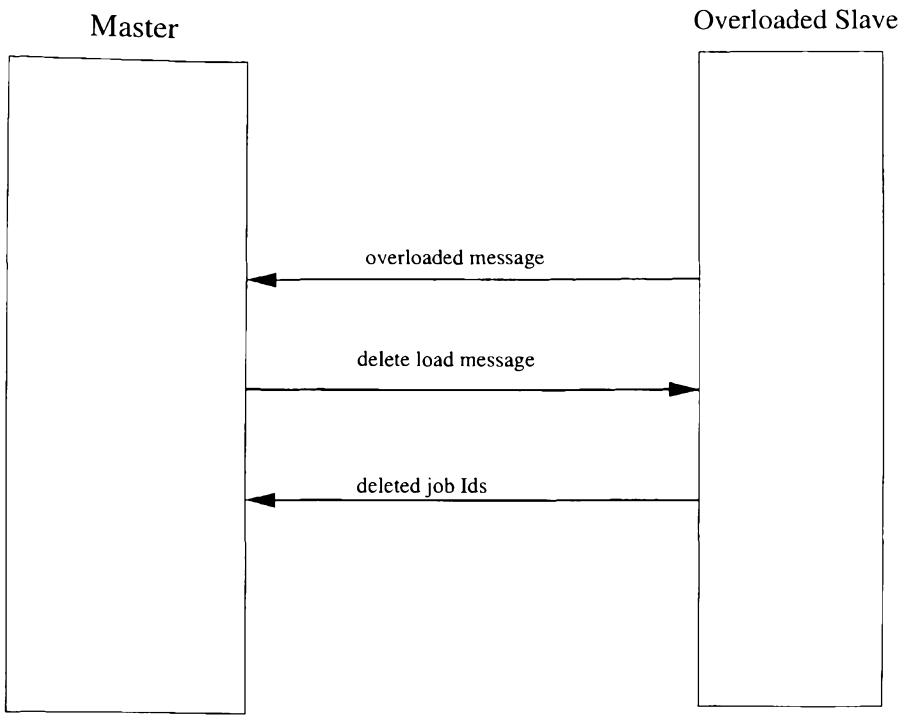

Figure 4. Slave Processor Overloaded

2.6.4 Slave Processor receiving additional load

Figure 5 below shows the communication protocol of the master distributing deleted jobs (received from the overloaded slave) to a lightly loaded slave. When the master processes a non-empty deleted job identifies data structure, it uses these identifiers as keys to obtain the jobs from its job repository and build up a queue of all the deleted jobs. The master now empties the deleted job identities data structure. The built deleted job queue is then send to the lightly loaded slave in the form of an add load message.

Upon receiving such a message the slave enqueues the jobs in the message to its job queue, and continuos processing jobs from the queue.

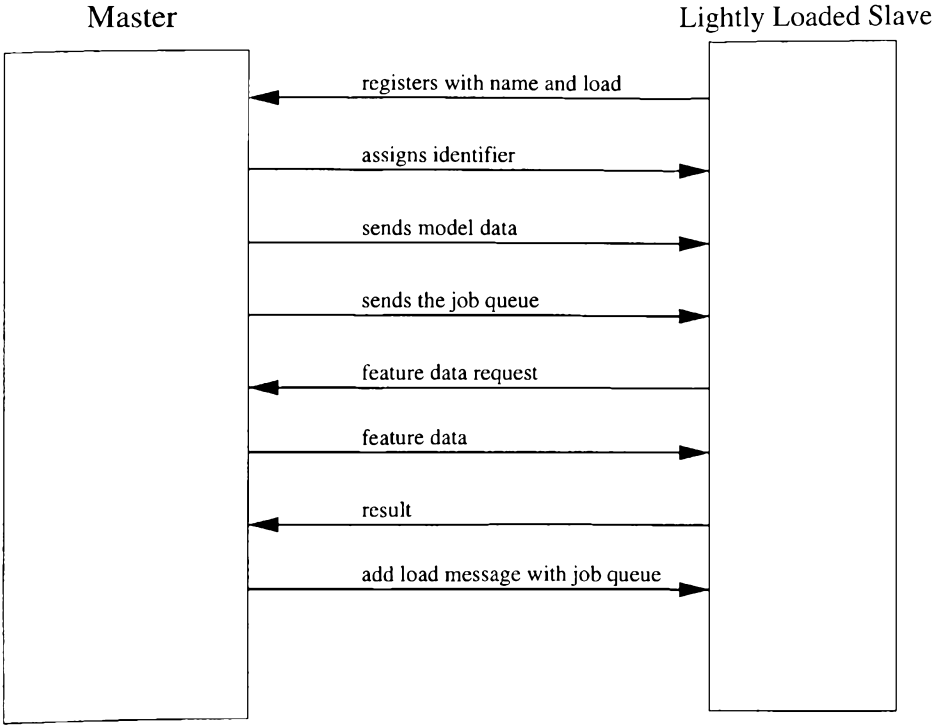

Figure 5. Slave Process receiving additional load

3. Algorithms

3.1 Overview

The proposed algorithm for the parallel speaker-recognition system is based on master-slave architecture. The existing serial system had a single Matlab process that called library C functions to calculate the HMM likelihood's. The testing process involves a number of utterances, which contain one or more tokens to be tested for HMM likelihood calculations against a number of models in the system. The algorithm starts by checking if the user has requested parallel or serial execution. If the user chooses not to run in parallel then the serial system is allowed to execute or else a master is created and the model data is registered on the master. Once the master is registered the Matlab process continues and determines the models that need to be tested against each of the tokens. Based on this information the jobs are created and registered on the master. Each job represents a token and a number of models that need to be tested against it. The job also requires other values associated with the token for the HMM likelihood computations. The Matlab process computes these values for each of the tokens that need to be tested and then the jobs on the master are updated with these values.

The master uses the load distribution algorithms to distribute the jobs. The various load distribution algorithms are discussed in the section 3.3. If the load of the processor on which the slave is running goes above a certain threshold, the master uses the load balancing algorithm to balance the system's work load. The load-balancing algorithm is discussed in the 3.4. The purpose of these algorithms is to speed up these calculating by

performing the HMM likelihood calculations in parallel on multiple processors, which would result in the improvement of over all throughput of the recognition system.

3.2 Assumptions

There are certain assumptions that are made regarding the system. These assumptions do not weaken the algorithms but are necessary for its successful working. The system has been implemented with the following assumptions:

1. The master assigns a unique identifier to each slave that has successfully of connected to it. The assignment of these identifiers is in sequentially increasing order.
2. The master and the slave communicate only by using a message object after initialization.
3. Messages between the master and the slave are never lost.
4. The master contains information about all the slaves but the slaves do not contain any information about each other.
5. A remote file copy is required to copy the slave file if the processor on which the slave is executing has a different file system from the master.

3.3 Load Distribution Algorithms

The load distribution algorithm used by the master is to distribute the jobs that have been registered by the Matlab process. Job queues are built for each of the slaves and sent over the socket for processing. The load distribution algorithms determine the number of jobs in each slave's queue. Three different algorithms have been evaluated.

3.3.1 Processor load based distribution

This algorithm uses the load of the processor on which the slave is executing. The number of jobs assigned is inversely proportional to the load. The load used is the average load of the processor over the past 5 minutes. This load figure corresponds to the average number of jobs in the run queue of the processor over the past 5 minutes. The number of jobs assigned to an i^{th} machine (M(i)) is given by the formula:

$$M(i)_{no\_of\_jobs} = ((\sum L(i) - L(i)) \ / \ \sum L(i))* no\_of\_jobs$$

Where L(i) is the load of the i^{th} machine.

3.3.2 Processor Clock speed based distribution

This algorithm uses the clock speed of the machine processors on which the slave is executing. The speed used is the clock frequency of the processors. Speeds of all processors of the machine are added and this value as a whole is taken as a basis for distribution. The number of jobs assigned is directly proportional to the value. This value

is termed as the processing power of the machine. The number of jobs assigned to an i^{th} machine is given by the formula:

$M(i)_{no\_of\_jobs} = M(i) / \sum M(i) * no\_of\_jobs$

Where M(i) is the processing power of the i^{th} machine.

3.3.3 Computation speed based distribution

This algorithm finds the number of floating point operations per second that can be performed by the processor. The number of jobs assigned is directly proportional to the number of floating point operations per second. This algorithm has not been implemented, as this information was not available with the system administrators.

3.4 Load-balancing Algorithms

The load-balancing algorithm used by this architecture is based on a sender-initiated load-balancing technique. When the slave load exceeds a certain threshold a message is sent to the master to inform its state. The master then performs the load-balancing by sending a message to the overloaded slave to delete some assigned jobs. The number of jobs deleted from the slave depends on the current load of the slave. The slave deletes the requested number of jobs and sends them to the master. The slave then proceeds with the processing of the remaining jobs. If the slave is highly overload the master sends a delete all jobs message and halts the slave. The deleted jobs are stored on the master. The master then assigns these to other slaves, which are not overloaded.

3.5 Theoretical Efficiency

The efficiency of the parallel speaker recognition system with respect to the conventional speaker recognition system with no parallelism can be evaluated by estimating the time taken by the parallel system. The various factors in evaluating the parallel speaker recognition system are:

- The time taken by the master to start and have the slaves connect and establish communication channel.
- The computation time taken by the Matlab process to convert the model data to Java objects.
- The communication time for the master to send model data to the slaves.
- The computation time taken by the Matlab process to convert the feature data to Java objects for all the phases for each test utterance.
- The computation time taken by the Matlab process to create the job table.
- The computation time taken by the master to distribute load using the load-balancing algorithm and to build the job queue.
- The communication time for the master to send job queues to the slaves.
- The computation time taken by the slaves to identify sub-jobs and data requirements.
- The communication time taken by the slaves to request uncached feature data and receives it.

- The communication time for the slaves to call the native C function using JNI and receive the likelihood scores.
- The communication time for the slaves to send the result to the master.
- The computation time taken by the master to group all results and make it available to the Matlab process in an appropriate format.
- Delays involved in starting and stopping communication for each new phase and test utterance.

The above points clearly illustrate that both extra communication and computation are required for a parallel speaker recognizer. For obtaining a performance improvement the time taken by the actual computation should be greater than the extra communication and computation. Communication over the sockets is expensive therefore the actual computation task given to the slave should be large enough to make this communication over head small. Therefore considerable thought has to be given to the size of jobs assigned to the slaves.

The efficiency of the parallel system is given by [8]:

$$\text{Efficiency} = \frac{\text{Execution time using one processor} * \text{processor clock speed}}{\text{Execution time using multiprocessor} * \sum \text{processor speed (i)}}$$

Where processor speed(i) is the speed of the i^{th} machine (slave) used

4. Results

4.1 Input Parameters

A number of tests are performed to evaluate the performance of the system. These tests are the basis of the proof of concept, which is to show that parallelism increases the throughput of a recognition system. The tests are first run in the serial mode on the fastest machine that will be used in the parallel tests. The results thus obtained are stored for later comparisons with the parallel output. Then the same tests are run in parallel on machines that are equal or slower in speed to the machine that was used for the serial testing.

Primarily three types of tests are performed. The first one uses Viterbi decoding of single-state 48 mixture HMMs, which were trained with 90 seconds of speech. The next two tests involve Viterbi integral decodes, one of them uses Error modeling and the other Asymptotic upper bound. These two methods differ in the procedure by which they find the integral decode around the observation point. The Error model is dependent of the feature data being tested, whereas the Asymptotic is independent of it. These two basic types of integral decodes are in use by the serial system to evaluate the integral decode around the observation point. The details of these methods are beyond the scope of this thesis. All the above tests are repeated for one, three and five second tokens. The token length corresponds to the length used to divide the 30 seconds speech sample into small chunks to obtain the tokens. Therefore we have different number of tokens for the same speech data with varying token length. The smaller the token length the greater the

computation required as the number of segments which can be obtained from the 30 second sample increase.

4.2 Expected Results

It is expected that the parallel test throughput is better than that of the serial. Also the throughput of the parallel tests must improve as the number of slaves increases upto a certain number of slaves, and deteriorate as the number of slaves are decreased.

The throughput of the parallel system is not always better than the serial throughput. For the parallel system to obtain better throughput the computation time of the jobs distributed by the parallel system should be greater than the communication time plus the extra computation time to execute the system in parallel. The greater the difference between the job computation time and the sum of communication and extra computation of the parallel system, the better throughput achieved. Thus a performance improvement is not expected for first kind of test that uses Viterbi decoding of single-state 48 mixture HMMs, as the communication time plus the extra computation time exceeds the computation time of the job itself. The performance improvement is expected in all the cases where integration is involved, irrespective of the method and utterance length used.

4.3 Functional and Reliability testing

These tests are performed for testing the parallel system for functionality correctness and reliability. The functionality correctness test includes the testing of the load balancing function of the system when one or more slaves entered the overloaded state and the load balancing function which distributes the load (jobs) among slaves. Forcing the machine of on which the slave is executing to the overload state by consuming its processor cycles by executing programs with infinite loops tests the load balancing functionality. This will increase the load on the machine, which in turn increase the load and send the slave into the overload-state. One or more slaves are made to enter the overload-state in the manner described above and the behavior of the system is analyzed. The load balancing algorithms are tested one at a time. Starting slaves on machines that are loaded at different levels to test the processor load based load-balancing algorithm. In a similar way starting slaves on varying clock speed processors tests the processor clock speed based load-balancing algorithm.

The stress testing are performed to analyze the reliability of the system. This test involves the killing one or more of the slaves explicitly by the user and observing the behavior of the system. Adding a new slave and restarting a killed slave is also done it this test.

4.4 Performance Improvement

The serial and parallel results below in tables 2, 3 & 4 show how well the parallel system has improved the performance. The first table shows the results from the serial

tests. The test identifier, machine name, elapsed time, number of speakers, token size and the method for the tests are specified for these serial tests. The token size corresponds to the length of the token utterance and the method specifies the type of test performed.

Test No.	Machine Name	Elapsed Time	No. Speakers	Token Size	Method
1.	Mozart	1min 26sec	9	5 sec	Viterbi decoding
2.	Mozart	1min 25sec	9	3 sec	Viterbi decoding
3.	Mozart	1min 27sec	9	1 sec	Viterbi decoding
4.	Mozart	4hrs 22mins	9	5sec	Error model
5.	Mozart	4hrs 29mins	9	3sec	Error model
6.	Mozart	4hrs 40mins	9	1sec	Error model
7.	Mozart	6hrs 19mins	9	5sec	Asymptotic
8.	Mozart	6hrs 28mins	9	3sec	Asymptotic
9.	Mozart	6hrs 36mins	9	1sec	Asymptotic

Table 2. Serial Test Results

The results of the same tests that are run in parallel are given in the parallel test results table with the same test identifier. In the parallel test results table additional fields for slave machine names, time gain with respect to the serial tests, efficiency and master machine name have been created. Efficiency has been calculated using the formula on page 33. The processor speeds are obtained from table 4, the list of available machine configurations. Fields, time gain and efficiency gives the performance improvement of the system with respect to the serial execution.

Test No.	No. of Slaves & Machine Name	Time Elapsed	Time Gain	Efficiency	Token Size	No. of Speakers & Method	Master
1.	2 Mozart 1 Weasel	16 mins 19 secs	-14 mins 53 secs	0.035	5sec	9, Viterbi decoding	Goliath
2.	2 Mozart 1 Weasel	17 mins 34 secs	-16 mins 09 secs	0.032	3sec	9, Viterbi decoding	Goliath
3.	2 Mozart 1 Weasel	61 mins 0 secs	-59 mins 33 secs	0.009	1sec	9, Viterbi decoding	Goliath
4.	2 Mozart 1 Weasel 1 Bach	1hrs 48mins	2hrs 34mins	0.82	5sec	9, Error model	Goliath
5.	2 Mozart 1 Weasel 1 Bach	2hrs 01mins	2hrs 28mins	0.75	3sec	9, Error model	Goliath
6.	2 Mozart 1 Weasel 1 Bach	2hrs 27mins	2hrs 13mins	0.65	1sec	9, Error model	Goliath
7.	2 Mozart 1 Weasel 1 Bach	2hrs 41mins	3hrs 35mins	0.80	5sec	9, Asymptotic	Goliath
8.	2 Mozart 1 Weasel 1 Bach	2hrs 45mins	3hrs 43mins	0.80	3sec	9, Asymptotic	Goliath
9.	2 Mozart 1 Weasel 1 Bach	3hrs 14mins	3hrs 42mins	0.69	1sec	9, Asymptotic	Goliath

Table 3. Speed Based Distribution - Parallel Test Results

Test No.	No. of Slaves & Machine Name	Time Elapsed	Time Gain	Efficiency	Token Size	No. of Speakers & Method	Master
1.	2 Mozart 1 Weasel 1 Bach	13 mins 59 secs	-12 mins 23 secs	0.040	5sec	9, Viterbi decoding	Goliath
2.	2 Mozart 1 Weasel 1 Bach	21mins 10 secs	-19 mins 45 secs	0.026	3sec	9, Viterbi decoding	Goliath
3.	2 Mozart 1 Weasel 1 Bach	52 mins 06 secs	-50 mins 39 secs	0.011	1sec	9, Viterbi decoding	Goliath
4.	2 Mozart 1 Weasel 1 Bach	2hrs 05mins	2hrs 21mins	0.71	5sec	9, Error model	Goliath
5.	2 Mozart 1 Weasel 1 Bach	2hrs 24mins	2hrs 05mins	0.63	3sec	9, Error model	Goliath
6.	2 Mozart 1 Weasel 1 Bach	2hrs 44mins	1hrs 56mins	0.58	1sec	9, Error model	Goliath
7.	2 Mozart 1 Weasel 1 Bach	3hrs 09mins	3hrs 10mins	0.68	5sec	9, Asymptotic	Goliath
8.	2 Mozart 1 Weasel 1 Bach	3hrs 50mins	2hrs 38mins	0.57	3sec	9, Asymptotic	Goliath
9.	2 Mozart 1 Weasel 1 Bach	4hrs 21mins	2hrs 35mins	0.52	1sec	9, Asymptotic	Goliath

Table 4. Load Based Distribution - Parallel Test Results

Index	Processor Name	No. of Processors	Processor's Speed
1.	Bach.cs.fiu.edu	1	300 MHz
2.	Dizzy.cs.fiu.edu	1	167 MHz
3.	Goliath.cs.fiu.edu	2	400 MHz, 400 MHz
4.	Grads.cs.fiu.edu	2	148 MHz, 148 MHz
5.	Mozart.cs.fiu.edu	2	750 MHz, 750 MHz
4.	Weasel.aul.fiu.edu	2	400 MHz, 400 MHz

Table 5. Available Machines Configuration

4.5 Analysis and Contributing Factors

Our results show that there is an improvement in throughput when error modeling and asymptotic methods of integral decode is being used. This proves that the concept of parallelism works for reducing the throughput of a speaker recognition system. These tables also show a decrease in the throughput when the Viterbi decoding with single-state 48 mixture HMMs was performed as expected.

The primary contributing factor for the improvement in the throughput time is clearly due to executing the jobs in parallel for both error model and asymptotic tests. The jobs associated with these tests needed large computation time. The performance improvement is due to the fact that this computation time exceeded the time for extra computation to run the system in parallel plus the communication time with the slaves.

The efficiency for these tests in most cases show that with 3 to 4 slave machines, less than 25% of the processing time is spend on communication and extra computation to run the system in parallel. Thus as a large fraction of the total processing time is used for processing the jobs (likelihood scores calculations) there is a good increase in throughput.

The tables also show that the efficiency obtained using the speed-based load distribution is better than the load- based distribution. This is due to the fact that, in load-based distribution the load gets distributed evenly when all the machines are lightly loaded. This results in larger number of jobs processed on lower speed machines.

There is a decrease in performance improvement in the case when Viterbi decoding with single-state 48 mixture HMMs is used because the jobs associated with it required less processing time. The computation time associated with these jobs was less than the communication time and extra computation time. The communication time per first phase remains constant irrespective to the type of method, thus this time had an adverse affect on the efficiency of the system in this case. This shows that an improvement in throughput can be achieved only when good amount of computations a can be given to the slaves. Both error modeling and asymptotic also use the Viterbi decoding, but along with integration. This makes the jobs associated with these methods have large computation time. Thus we see an increase in performance.

4.6 Reliability and Robustness

This system has a high degree of robustness with respect to the slave side, but is less a low robust at the master side. The events of slaves crashing and recovering

are handled gracefully. The system can stand most of the unexpected interrupts on the slave side. But in the event of the master crash the system does not recover. As the master and the Matlab process runs on the same processor, the recovery of the master in the event of a crash is not of any help as the Matlab process itself dies. This is clearly due to the fact that the Matlab process runs on a single processor and is not an architectural flaw. The architecture of the system has the normal limitations of slave master architecture – the centralized control that is not reliable.

Tests were performed for measuring the reliability of the system. The system was able to recover from the explicitly killed slaves. The jobs of the killed slave were assigned to other running slaves in the system. In the event when the slave was restarted the master accepted the slave and assigned it new jobs. Thus the reliability of the system with respect to the slaves side has been proved to be good.

5. Conclusions

5.1 Contributions

Parallelism has not been used till date to increase the throughput of a speaker recognition system. This concept of the parallel speaker recognizer proves that parallelism can be used to increase the throughput of the recognizer. This thesis is a step towards large-scale speaker recognition system which does the recognition process on multiple speakers in parallel. Such large-scale speaker recognition systems have many applications, some of them being an automated telephone call bank, a credit card verification process done by the merchant over the telephone and many more of such kind. The large-scale systems cannot be dependent on good pruning techniques for an optimum throughput as multiple speaker use the system at the same time, processing one speaker a time will delay the processing of other speakers in queue. Parallelism seems to be one of the better ways and this has been proposed here with this thesis. The results obtained can be used to compare performance of other techniques.

5.2 Future Work

As mentioned in section 5.1, this thesis is a step towards the large-scale speaker recognizer. The primary task of the future work would be to extend the system to handle more than one recognition process at any given time, and processing these tasks in parallel.

Other future tasks are to optimize the existing system. The processing of the message form the slaves can be handled at the connection handler's level that would avoid the bottleneck at the master. The slaves can be kept busy all the time by populating there job queues as soon as they are empty, to avoid the loss of computation time. Jobs of only one utterance are distributed at a time and the Matlab process waits until all the results of the jobs return, as of the present design. A better throughput could be achieved by distributing jobs of more than one utterance at a time. This would result in larger jobs to be distributed at once, which decreases the communication overhead. More sophisticated load distributing and load balancing techniques can be used distribute the jobs to use the better available computing processors. The system can be decentralized and made more reliable by having multiple masters.

List of References

[1] Q. Lin, E-E. Jan & J. Flanagam (1994) "Microphone arrays and speaker identification," IEEE Trans. Speech & Audio Proc. 2, 622-628.

[2] S. Kwong and C. W. Chau (1997) "Analysis of parallel genetic algorithms on hmm based speech recognition system," IEEE transactions on consumer electronics. Vol. 43, 1229-1233.

[3] Carl D. Mitchell, Mary P. Harper, Leah H. Jamieson (1995) "A parallel implementation of a hidden Markov model with duration modeling for speech recognition," Digital Signal Processing. Vol. 5, 43-57.

[4] Mine R., Kobayashi T., Shirai K. (1996) "Speech recognition in non-stationary noise based on parallel HMMs and spectral subtraction," Systems and Computers in Japan. Vol. 27(14), 37-44.

[5] Noda H., Shirazi MN., Zhang B. (1994) "A parallel-processing algorithm for speech recognition using Markov random-fields (MRF)," Systems and Computers in Japan. Vol. 25(2), 92-100.

[6] Rabiner L., Juang BH. (1993) - Fundamentals of speech recognition, ISBN 0-13-015157-2, Prentice Hall Inc.

[7] Roch, Marie and Lee Chin (1995) personal conversation.

[8] Barry Wilkinson, Michael Allen (1999) - Parallel Programming techiniques and applications using networked work stations and parallel computers, ISBN 0-13-671710-1, Prentice Hall Inc.

[9] Speech Recognition Using Hidden Markov Models,

http://www.cnel.ufl.edu/~yadu/hmm.html, (2000) University of Florida.

[10] ITT/Linguistic Data Consortium, (1992). King kingdb.doc and collectn.doc files. CD-ROM.

[11] Roch, Marie (2000) "Robust Hidden Markov Model classification schemes for speaker recognition using integral decode", Doctoral thesis.

Unpublished References

[12] Roch, Marie and Hurtig, Richard (2001) "Optimization Techniques for the Integral Decode," submitted to IEEE Transactions on Acoustics, Speech, and Signal Processing.