Electronic Research Archive of Blekinge Institute of Technology
<http://www.bth.se/fou/>

This is an author produced version of a conference paper. The paper has been peer-reviewed but may not include the final publisher proof-corrections or pagination of the proceedings.

Citation for the published Conference paper:

Title:

Author:

Conference Name:

Conference Year:

Conference Location:

Access to the published version may require subscription.

Published with permission from:

Performance Evaluation and Prediction of Open Source Speech Engine on Multicore Processors

Christine Niyizamwiitira
School of Computing
Blekinge Institute of Technology
37179 Karlskrona, Sweden
+46 455385861
cnw@bth.se

Lars Lundberg
School of Computing
Blekinge Institute of Technology
37179 Karlskrona, Sweden
+46 455385833
llu@bth.se

ABSTRACT

This paper quantifies the performance of the core part of voice driven web using free and open source speech engine; the speech engine which is very high computation demanding, it consists of Automatic Speech Recognition (ASR) and Text To Speech (TTS). Two open source programs, Sphinx-4 and FreeTTS-1.2.2 are used for ASR and TTS respectively. These two programs are executed on 2 different hardware multicore processors with 4 hyperthreaded cores, and 8 cores respectively. The response time with respect to the load variance and the number of cores is measured and predicted using a linear regression model.

The results show that, the response time is linear with respect to the input length, this property can be used to directly predict the response for any input length. Moreover, though the response time and the speed up increases as the number of cores increases, the regression coefficients and number of threads reveal that ASR benefits from multicore. The speedup factor for ASR is 1.56 for 8 cores. However for FreeTTS, though being sequential the speed up from the program itself is insignificant, there is about 1.43 speedup for 8 cores, that comes from the system's contribution. Our findings show that the generalization of the results for multicore processor does not apply to hyperthreading. This paper presents the investigation that is useful for educators, researchers, and applications' developer in voice based applications' domain.

Categories and Subject Descriptors

D.4.8 [Performance]: Modeling and Prediction

General Terms

Performance, Measurement, Experimentation.

Keywords

Speech recognition, Text to speech, Voice driven web, Multicore performance, Performance prediction, linear regression, Open Source.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

MEDES'13, October 28-31, 2013, Neumünster Abbey, Luxembourg.
Copyright © 2013 ACM 978-1-4503-2004-7/10/10...\$10.00.

1. INTRODUCTION

Nowadays voice based-application are emerging very fast, there exists many voice-based application from different vendors that are not open to researchers. Therefore it becomes difficult to know what is behind scene. For that reason, we investigate free and open source voice engines in order to better understand voice application's phenomenon. Generally, there are three main architectures for Voice driven web; the first architecture uses prerecording to render the information, and the server plays back the recordings. The interaction between server and terminal is done through Dual Tone Multi Frequency (DTMF) and/or ASR (Automatic Speech Recognition). This is a user friendly approach since one uses recordings from humans. However, this architecture is not suitable when the information is updated frequently due to almost impossible and tedious recording work [1].

The second architecture consists of ASR as well as TTS (Text To Speech) that are implemented on the server side. Where, the server performs all speech processing; this server-based processing permits fast processing and consequently fast response time. This architecture is scalable because it simplifies the introduction of new components such as hardware resources, adding new services, and applying load balancing thus putting fewer requirements on the terminals

The third architecture performs all of the ASR and TTS processing on the terminal instead of server side and the traffic between the server and the terminal are only text messages over the network instead of voice as it is for the previous architectures. This architecture is limited to smart phones that can handle ASR and TTS, and it requires installations of ASR and TTS on the terminal. ASR and TTS are by nature computational demanding; hence when they are implemented on the handheld device the performance decreases due to limited computational capability of the latter. The architecture tradeoffs show that the architecture that implements ASR and TTS on the server side has better performance [2].

Voice driven web should provide real-time communication and handle many client requests simultaneously. Therefore, it requires huge processing resources. For this reason, the second architecture which is server-based speech processing has been found promising if it is implemented in an appropriate server environment.

The speech engine is highly computational demanding, and multicore processors are necessary. As far as this paper is concerned, the performance of speech engine is studied where ASR and TTS are executed on multicore processors with shared memory. In this paper, the performance metric is the response time of ASR and TTS. We will measure the response time on two different multicore processors. Based on these measurements we will build a performance prediction model. The predicted response time as a function of the input load length is derived according to linear regression model; the response time as a function of the number of cores is also investigated. Voice driven web like any other voice application can be developed using open source ASR and TTS. In this paper, we use free and open source, Sphinx-4 and FreeTTS-1.2.2. The rest of the paper is structured as follows; the second section presents background, motivations, implications of this study, and related work. The third section clarifies the research methodology. The fourth section presents results. The fifth section discusses results and their contributions; and study validity. The sixth section draws conclusion and provides recommendations.

2. BACKGROUND

2.1 Concept overview

Voice driven web is a Voice XML based application that provides automated interaction condition for callers to retrieve information from web through telephone keypad or speech recognition. The architecture of voice driven web has three main parts; (1) a voice browser that interact directly with the terminal, (2) a speech technology server, i.e., ASR that recognize the input speech from the end-user and TTS that reads the content of the web in response to the end-user request, and (3) the back end which is composed of a web proxy and the World Wide Web (www). ASR and TTS are the most computationally demanding parts in voice driven web. Hence, implementing this part on powerful processing hardware would improve the overall response time which is a crucial factor for the quality of voice driven web systems.

In 2008, there existed two and four cores machines, some experts believe that by 2017 embedded processors could support 4,096 cores, server CPUs might have 512 cores and desktop chips could use 128 cores [5]. To select the appropriate hardware architecture, performance evaluation of different candidates is vital. In this paper, the performance of ASR/TTS on multicore processor architectures is evaluated. We also attempt to predict the performance for a large number of cores. The performance is evaluated as the response time. The response time for ASR is the time taken to recognize a certain audio input until we get the text output, whereas for TTS, it is the time taken to get audio output from a certain text input.

In voice driven web, the response time is influenced by network/communication delays, and speech processing delays. In this paper, we focus on evaluating speech processing. If we assume that the network is good enough to carry the voice, the response time of the voice driven web can be estimated based on the ASR and TTS response times. We use open source speech recognition named Sphinx-4 [6]. It is a state-of-the-art continuous-speech, and speaker-independent systems based on hidden Markov (HMM) decoded using the frame synchronous Viterbi algorithm [7]. It has also N-gram language models. Sphinx-4 is distributed under an

academic BSD-style license. The code and binaries are free for commercial and non-commercial use. We also use open source text to speech named FreeTTS. It is based on CMU's Flite, which is derived from the Festival and the FestVox project, from Carnegie Mellon University [3]. FreeTTS is also released under a BSD license. The ideas behind choosing Sphinx-4 and FreeTTS to conduct this study are; (1) portability because they are portable entirely written in Java and are flexible, adjustable to the extent that they can be edited and configured to fit the user requirements, (2) the software is free and easily available. The investigation of the speech engine as open source on multicore processor helps the researchers and developers to adapt to new hardware technology.

2.2 Related work

In [8], the authors studied fine-grained application concurrency in a HMM-based inference engine for large vocabulary continuous speech recognition (LVCSR). They measured the performance based on Intel core i-7 and GTX280 NVIDIA processors. Sphinx-4 that also uses HMM and Viterbi algorithm is studied.

In [9], Yoshizawa et al. proposed a scalable architecture for real-time speech recognizers based on word hidden Markov models (HMMs). This architecture effectively uses parallel computations on the word HMM structure.

A performance model for multicore environments has been studied in [10] where Chen et al. presented an online technique for estimating the performance and the power consumption of interacting processes in multi-programmed and multicore environments. Willie et al. compared the performance of FreeTTS which is implemented in Java, and C. They used a dual core processor for their experimentation. The authors found that FreeTTS that is developed in Java performs better than in C in terms of fast response time and algorithm modifications becomes easier [11]. The same authors studied the performance of Sphinx-4, they explained the framework and properties of Sphinx-4, and they presented preliminary performance measures on a dual core processor [6]. In [12], Adve came up with a simple deterministic model for parallel program performance prediction, using deterministic values to represent mean task times including communication. This model can be used to quantify and understand program performance, and to predict the impact of system and program changes. In [13], the authors studied how to predict the performance of multicore on Solaris containers, using linear regression. They come up with optimistic results where the correlation between response time and the number of CPUs and number of threads is linear in logarithmic space.

Across literature, the performance prediction of ASR and TTS for multicore processor with high number of cores has not been studied. Sphinx-4 and FreeTTS were particularly studied on dual core processor. Whereas in this paper, we study the performance of those on dual quad core and Intel core i-7. We provide performance evaluation of the speech engine, i.e., ASR and TTS. We measure the response time, the speedup, the CPU utilization, and number of threads. The measurement is conducted on different number of cores are done on different hardware. Moreover, the response time as a function of input load is measured and prediction is derived based on linear regression.

3. METHODOLOGY

3.1 Experiment setup

In the experiment, we use CMU Sphinx-4 automatic speech recognition, which is an open source speech recognizer developed

by Carnegie Melon University (CMU) [14]. Sphinx-4 has two packages, the bin which is enough for creating application, and the source which is modifiable and containing the test part to fix bugs after modifying the code. The packages are downloadable under sourceforge repository, statistics shows that 3285 downloads are done until 23rd of August 2013.

We also use FreeTTS 1.2.2 which is also an open source text to speech which is built on CMU's flite, FreeTTS has three downloadable packages which are bin, source and test [4] [12]. The bin package is sufficient for creating application, but if you like to modify code then source is the correct package to use. If it happens to make change to code, test package will help to fix bugs and regression that might introduced during changes. Downloads under source forge repository show that 438 downloads have been done until 26th April, 2013. The statistics for both FreeTTS and Sphinx-4 show how much these open sources are popular in the speech application domain.

The following equipment is used in this experiment: (1) Intel Core i-7-2620M / 2.7 GHz, 8GB RAM, dual-core with hyperthreading of two threads per core, i.e., 4 logical cores. It runs the Ubuntu operating system 12.10 LTS, 32 bits. (2) Intel(R) Xeon(R) E5335 @ 2.00GHz dual quad core, 16 GB RAM, with 8 cores without hyperthreading, which runs Ubuntu 12.10 LTS, 64 bits.

We isolated the ASR/TTS servers and measured the response time for different loads. The CPU utilization and the number of threads are also measured. In this experiment, the input to the ASR is a continuous audio file with fifteen spoken digits during 8 seconds; the utterances are separated by silences [15]. This audio has similar characteristics as a typical speech from a user; this audio is recorded in a native speaker voice. The text that is used in this experiment, is the first five chapters of Alice's Adventures in Wonderland by Lewis Carroll [16]. It is a text of 70 minutes. We have chosen this text, because it was first used by the pioneer of FreeTTS for testing.

The initial length for FreeTTS is the first five chapters of Alice Adventure, it is about 10900 words. For a matter of varying the length of input load, we linearly concatenated up to 9 times the initial. The highest load is $10900 \times 9 = 98100$ words.

The initial audio load for ASR is 80 seconds audio length. We vary the load by concatenating initial load until 9 times, i.e., the highest load is $80 \times 9 = 720$ seconds. At every increase of load we run it 100 times in order to measure the average response time.

We use the Jvisualvm profiling tool for examining CPUs utilization, memory consumption and thread of the application [17].

3.2 ASR and TTS architecture

Figure 1 shows ASR recognition that is used in the experiment. ASR main parts are feature extractor, hidden Markov and Viterbi decoding. Viterbi decoding is the most computation demanding part. The decoding uses the frame synchronous Viterbi algorithm, which is described in Figure 2. An example of this decoding is described in Figure 3, where the word "one" is decoded.

In Figure 3, for each state with more than one connection, Viterbi scores the state with the probability of the most probable connection. Then it keeps a backpointer to which state it is connected to. For example at time $t=3$, 0.06 is selected as the maximum probability the rest is pruned away. The pruning process follow beamwidth threshold, and the garbage collector helps in that, i.e., the pruner removes the terminal token and identifies the token and any unshared tokens that are unused.

Figure 1. Architecture of Speech recognition processing steps.

Figure 2. Connected word Viterbi Search.

Figure 3. Example of Frame synchronous Viterbi decoding

For a word "One".

Then the garbage collector reclaims the associated memory [6]. Threads are used to score each state probability at each time frame. Each core executes a thread. The size of the thread pool for scoring states is set dynamically according to the number of available cores. The processing of TTS is done in sequence with the following main steps; text normalization, linguistic analysis, lexical analysis, prosody generation, speech synthesis [11].

3.3 Linear regression Performance Modeling

As described in the previous section, the maximum number of cores considered in the experiment is 8 cores. Based on the experimental results, we want to predict the performance of ASR/TTS for higher input load and higher number of cores. To do that, we have to apply a fitting function to the experimental results

and extend the pattern for further input value. The linear regression is a well-known simple fitting function [18]. Therefore we use linear regression to extrapolate the experimental results. Linear regression fits a straight line through data to determine the slope and intercept with the y -axis. Using the linear regression method we get the least square regression line (LSRL) that is given in Equation 1 [19]. In the context of this paper, LSRL helps us to predict the response time with respect to input length.

$$f_i(x) = a_i + b_i x \quad (1)$$

In this case, $f(x)$ is a dependent variable that represents the response time that is predicted based on the independent variable x which represents the input length; a is a regression constant and b is the regression coefficient and it is also the average change in the dependent variable for a unit change in the independent variable. The index i represents the number of cores.

Let $f(x) = \hat{y}$, then Equation 1 becomes Equation 2.

$$\hat{y} = a_i + b_i x \quad (2)$$

$$a_i = \frac{\sum[(x_i - \bar{x})(y_i - \bar{y})]}{\sum[(x_i - \bar{x})^2]} \quad (3)$$

$$b_i = \bar{y} - (a_i * \bar{x}) \quad (4)$$

Here, y_i is the measured response time on i cores.

After finding the regression equation, the coefficient of determination that helps to check how well this equation fits the data is given in Equation 5. The coefficient of determination R^2 ranges from 0 to 1; as its value increases as the dependent variable can be better predictable from the independent variable. $R^2 = 1$ represents perfect correlation between the variables in question, while $R^2 = 0$ represents no correlation [19].

$$R^2 = \{(\frac{1}{n}) * \sum[(x_i - \bar{x})(y_i - \bar{y})] / (\sigma_x * \sigma_y)\}^2 \quad (5)$$

Besides R^2 , the Mean Squared Error (MSE) is also used to measure the precision between measured and predicted values based on regression method. When the measured and predicted values are more strongly correlated, MSE tends to approach zero.

$$MSE = \frac{1}{n} \sum_{i=1}^n (\hat{y}_i - y_i)^2 \quad (6)$$

Where n is the number of cores, $\hat{y}$ is the predicted response time.

Figure 4. Performance prediction model.

4. RESULTS

Figure 4 (i) shows an overview of the model; the inputs are different load length, number of cores, and the ASR/TTS case, the output from the model is the predicted response time. Equation 2 represents the output; and it is composed of two components a and $b x$ that contribute to the response time and the speed. The response time is predicted in two dimensions. Figure 4 (ii) shows a matrix which would contain the response time in two dimensions; vertically, we have the response time as a function of the input length, and horizontally we have the response time as a function of the number of cores. Scenario 1 is the experiment on Intel xeon dual quadcore. Tables 1 and 2 show the regression parameters values for ASR and TTS respectively. Figures 5(i, ii) show the response time for ASR and TTS, it is the time that it takes to get the audio output from a text. The response time for ASR, is the time that it takes to get the text as output of an audio file. The figures show also the linear regression fitting line with respect to measured data; these lines are the predicted response times. Scenario 2 uses the same procedure as Scenario 1 on Intel core i-7. Tables 3 and 4 show the regression parameters' values. Figures 5(iii, iv) show the response time and the linear regression fitting line for ASR and TTS respectively. The regressions parameters' values are shown in Tables 1, 2, 3, and 4.

Table 1. Regression parameter's values for ASR on 8 CPU

	1C	2C	3C	4C	5C	6C	7C	8C
a	7.77	5.56	4.93	4.80	4.71	4.68	4.50	4.50
b	1.28	1.18	1.10	0.97	1.06	1.08	0.92	0.83
R²	0.98	0.99	0.98	0.99	0.98	0.98	0.99	0.99
MSE	0.10	0.02	0.02	0.01	0.10	0.10	0.11	0.10

Table 2. Regression parameter's values for TTS on 8 CPU

	1C	2C	3C	4C	5C	6C	7C	8C
a	4.29	2.48	1.68	1.68	1.49	1.38	1.42	1.33
b	4.54	3.62	3.47	3.47	3.39	3.40	3.38	3.33
R²	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.99
MSE	0.14	0.17	0.09	0.11	0.09	0.17	0.19	0.17

Table 3. Regression of TTS on the 4 cores CPU

	1C	2C	3C	4C
a	3.37	1.14	1.76	1.16
b	3.53	2.82	2.81	2.80
R²	0.99	0.99	0.99	0.99
MSE	0.14	0.11	0.32	0.21

Table 4. Regression on ASR on the 4 cores CPU

	1C	2C	3C	4C
a	3.65	3.62	2.35	2.35
b	0.63	0.71	1.76	1.66
R²	0.96	0.98	0.95	0.95
MSE	0.07	0.02	0.01	0.01

Figure 6 and 7 are the results from JvisualVM for the ASR and TTS respectively. Figure 6 shows an example of typical ASR parallel scoring threads when 4 cores are enabled. We can see that the self-time of the concurrent method, i.e., java concurrent .Future Task which is composed of 4 parallel ThreadedAcousticScorer threads consume 100% of the CPU time that is used to complete the total work. Figure 7 shows a TTS example of threads' repartition according to CPU time consumption. The main thread handles all the work in a sequential manner. Under hot spot method, we see that, the IdentityHashMap method takes the highest percentage of CPU time.

Initially, when we run on a single core, we have 11 threads in TTS, 10 of those are for java virtual machine (JVM) functionality, i.e., they are not part of TTS program. Attach Listener, Thread-1 for audio output, Service Thread, C2 CompilerThread1, C2 CompilerThread0, Signal Dispatcher, Finalizer, Reference Handler, VM thread, VM periodic thread. Those threads do not consume significant CPU time. Another thread is the main thread and it is the actual thread of TTS.

It sequentially executes the program code. As the number of cores increase, we get additional threads that run in parallel; those are garbage collectors which are inherently parallel. Garbage collector contributes very little to the speed up of the program. The total threads number follows the Equation 7.

$$x_i = x_{i=1} + (\#cpus) \quad (7)$$

Where x is the total number of thread and i is CPUs number

Initially, ASR runs on a single core with 9 threads for java virtual machine (JVM) functionality; Attach Listener, Service Thread, C2 CompilerThread1, C2 CompilerThread0, Signal Dispatcher, Finalizer, Reference Handler, VM thread, VM periodic thread. and the main thread that execute the whole functions of ASR. The main thread executes sequentially for a single core.

However, as the number of cores increases; the total number of threads increases according to Equation 8. There is one scoring thread increase and one thread garbage collector increase per one core increase. For example; on 4 cores we have $10+8=18$ threads. Whereas on 8 cores we have $10+16=26$.

$$x_i = x_{i=1} + 2(\#cpus) \quad (8)$$

Where x is total number of thread and i is number of CPUs.

Figure 6 shows an example of typical parallel scoring threads when 4 cores are enabled, how many times the threads have been invoked during the runtime. If we take a look in table 3 and 4, the "b" values, the relative speed up of the response time on Intel core i-7 is not following any trends, hence not easy to predict with respect to number of cores.

Figure 5. Measured and predicted response time for ASR and TTS.

Therefore we consider the relative speed up measured on Intel Xeon dual quad core where some conclusions can be drawn. The speed up measurements shows trends that could be predicted. In Table 6 and 7, at every input length, the relative speed up is normalized in a sense that the response time for one enabled core is said to be 1. This response time is divided by the response time for each more enabled cores. Values in Table 6 and 7 are given by equation 9. The response time for one core is expected to be the highest and the one for the 8 cores is expected to be the smallest.

$$\frac{(Li, 1C)}{(Li, iC)} \quad (9)$$

iC is number of enabled cores; Li is input length.

Table 5 shows the number of threads for ASR and TTS with respect to number of cores. The CPU utilization on Intel-xeon is given in Figure 8. As a matter of representation, the CPU utilization shown in Figure 8 is for the smallest and for the biggest input length. For both ASR and TTS, the CPU utilization does not change significantly for different input load length on the same number of cores. Instead it decreases as the number cores increases (see Figure 8).

Call Tree - Method	Time [%]	Time	Invocations
ThreadedAcousticScorer-1-thread-4		1536 ms (100%)	1
ThreadedAcousticScorer-1-thread-2		1531 ms (100%)	1
ThreadedAcousticScorer-1-thread-3		1509 ms (100%)	1
ThreadedAcousticScorer-1-thread-1		1496 ms (100%)	1
Thread-0		0.161 ms (100%)	1
DestroyJavaVM		0.000 ms (0%)	1

Hot Spots - Method	Self time [%]	Self time	Invocations
java.util.concurrent.FutureTask.run ()		6085 ms (100.2%)	50507
java.util.logging.LogManager\$Cleaner.run ()		0.162 ms (0%)	1
java.lang.ApplicationShutdownHooks\$1.run ()		0.000 ms (0%)	1

Figure 6. ASR threads.

Call Tree - Method	Time [%]	Time	Invocations
Thread-0		5.21 ms (100%)	1
main		3.15 ms (100%)	1

Hot Spots - Method	Self time [%]	Self time	Invocations
java.util.IdentityHashMap\$KeySet.iterator ()		0.936 ms (11.2%)	2
java.util.IdentityHashMap.keySet ()		0.752 ms (9%)	1
java.util.logging.LogManager.reset ()		0.543 ms (6.5%)	1
java.lang.Thread.start ()		0.320 ms (3.8%)	2
java.util.logging.LogManager.resetLogger (java.util....)		0.303 ms (3.6%)	36
java.util.Hashtable.get (Object)		0.258 ms (3.1%)	36
java.util.Hashtable.<init> (int, float)		0.220 ms (2.6%)	1
java.util.logging.Logger.setLevel (java.util.logging....)		0.216 ms (2.6%)	36
java.util.logging.LogManager\$LoggerContext.findLog...		0.206 ms (2.5%)	36
java.lang.ApplicationShutdownHooks.runHooks ()		0.196 ms (2.3%)	1
java.util.logging.LogManager.checkPermission ()		0.193 ms (2.3%)	37

Figure 7. TTS threads.

Table 5. Number of Threads

#cores=>	1	2	3	4	5	6	7	8
ASR	10	14	16	18	20	22	24	26
TTS	11	13	14	15	16	17	18	19

5. DISCUSSIONS

In ASR, the scoring of Viterbi algorithm is done in parallel, where at each time frame each HMM state is scored by a thread. The threads are generated with respect to the number of enabled cores

Table 6. ASR Normalized speedup

	1C	2C	3C	4C	5C	6C	7C	8C
L1	1.00	1.26	1.39	1.50	1.48	1.49	1.54	1.60
L2	1.00	1.30	1.44	1.58	1.54	1.53	1.58	1.66
L3	1.00	1.27	1.47	1.54	1.53	1.50	1.55	1.63
L4	1.00	1.24	1.46	1.48	1.44	1.45	1.52	1.57
L5	1.00	1.17	1.32	1.47	1.40	1.43	1.49	1.57
L6	1.00	1.16	1.37	1.43	1.40	1.34	1.51	1.50
L7	1.00	1.14	1.29	1.42	1.38	1.36	1.43	1.50
L8	1.00	1.13	1.31	1.43	1.37	1.37	1.44	1.52
L9	1.00	1.14	1.29	1.43	1.33	1.33	1.41	1.50

Table 7. TTS Normalized speedup

	1C	2C	3C	4C	5C	6C	7C	8C
L1	1.00	1.43	1.65	1.70	1.72	1.73	1.73	1.74
L2	1.00	1.34	1.57	1.61	1.64	1.64	1.65	1.68
L3	1.00	1.37	1.52	1.54	1.55	1.58	1.59	1.64
L4	1.00	1.33	1.47	1.53	1.53	1.53	1.51	1.54
L5	1.00	1.34	1.43	1.47	1.47	1.49	1.49	1.51
L6	1.00	1.30	1.36	1.42	1.44	1.43	1.44	1.46
L7	1.00	1.28	1.34	1.38	1.39	1.38	1.41	1.46
L8	1.00	1.26	1.41	1.43	1.44	1.44	1.45	1.46
L9	1.00	1.31	1.38	1.38	1.41	1.41	1.41	1.43

CPU utilization on Intel Xeon

Figure 8. CPU utilization for ASR and TTS.

(see Table 5). Moreover, the garbage collector which is important during pruning where it reclaims the memory at each path pruned out is also processed in parallel with all available cores. In TTS, all steps are processed in sequence. However, the garbage collector that helps to manage the memory is processed in parallel with all available cores. This garbage collector contributes almost 2 % to the total work. Figures 5 (i, ii, iii, iv) show that the response time for both ASR and TTS is linear as a function of the input length.

The experiment conducted on Intel core i-7 processor shows that when we enable 2 cores from different physical cores the execution time significantly decreases compare to enabling two

hyperthreads on the same core. Keeping in mind that we have two physical cores and 2 hyper threads per each, this makes 4 logical cores on this processor. However, when we enable two cores that belong to same physical core, the execution time is similar as if only one core is enabled. Consequently, when three cores and four cores are enabled, we get the same execution time as if only 2 physical cores are enabled. In the worst case as observed in Figure 5 (i), when three and four cores are enabled, this adds overhead; hence introduce more delay to the response time. We observe that the execution time for ASR on 4 cores is higher than even on a single core. This is due to contention of shared resources when the parallel part tries to execute on separate logical cores in parallel. It is seen that hyperthreads share resources, for Intel core i-7, L1 cache and L2 cache are shared by hyperthreads that belong to the same physical core.

Furthermore, hyperthreads that belong to different physical cores share L3 cache. The speed up as shown by the slope value “b” in Table 4, decreases for almost a half from one core to four cores. The hyperthreading does not affect TTS because TTS executes sequentially, the execution time seems to be almost the same no matter how many cores or hyperthread are enabled. The value of “b” in Table 3, shows that the average speedup is about 26 %, i.e., for 3.37 seconds on one logical core to 2.80 seconds for four logical cores. This is Amdahl’s law, whereby the speed up of a program depends on how much parallel it is [20]. The part that increases threads is the garbage collector and this count almost 1.7 % to 2 percent of the whole work to be done. In this paper, neither ASR, though it has some parallel part, nor TTS benefits from hyperthreading. The main performance bottleneck is the access to the memory, whereby two virtual cores, i.e. hyperthreads contend to share L1 cache of 32 KB. Therefore this prevents taking advantage of enabling hyperthreading for these particular applications.

The results of ASR from Intel-xeon dual quadcore, shows that the number of threads that are generated, follows the number of enabled cores. This is shown in Table 5. The slope of the response time is shown in Table 1 and in Figure 5(ii). The slope (b value) shows the average speed up, it is about 1.54, from 1.28 to 0.83 for one and eight enabled cores respectively. Table 6 shows the normalized speed up, where we have the average of 1.56 for eight enabled cores. Both results from regression model and normalization tell us that the speedup is generally about 1.55, if we average both results. Consider the number of threads for ASR in Table 5 and the slope “b” in Table 1. At one core increase we see two threads increase, the garbage collector that counts for 2 to 3 percentage of the total work. The Scoring threads contribute to the speed up significantly. However, the speedup does not grow that fast as expected instead it slowly increases but gradually. There is about 7 % speed up at each core increase.

Though we cannot give strict conclusion based on CPU utilization only. It can help us to investigate and understand more the behavior of our programs. The CPU utilization in Figure 8 decreases from 100 % to 31% from 1 core to 8 cores shows that ASR consumes for example 31 % of 8 available cores. Additionally, the average speed up is from both regression and normalization is about 1.55 for 8 cores. Therefore, we can generalize our performance model to more cores; i.e., the right part of the matrix in Figure 4(ii). Increasing one core corresponds to 7 % speed up increase.

Moreover, based on the fitting line it is straightforward to predict the response time on the basis of exclusively multicore processors with hyperthreading disable or without it at all. This can be done for any input length for ASR when the number of cores varies from 1 up to 8. This corresponds to the lower left part of the matrix under the shaded part in Figure 4 (ii).

About TTS, the speed up as the slope “b” shows in Table 2 is averagely 1.36, from 4.54 to 3.33 for one and eight enabled cores respectively. Table 7 shows the normalized relative speed up that stays constant as the number of cores increases. The relative normalized speed up is about 1.50. The speed up from regression and from normalization is averaged to 1.43. The CPU utilization that goes from 100% for one core to 17% for 8 cores, i.e., cores are not exploited as much as ASR does because ASR has a parallel part. However like ASR, as Figures 5(iii, iv) shows, the execution time for TTS increases linearly as the input length increases.

Comparing ASR and TTS, even though ASR has a parallel part, the speed up is slow. Obviously, as expected the speed up is better than for TTS. However, TTS also has a small speed up increase, i.e., OS, java virtual machine threads and the garbage collector contribute to that speed up. To validate the study, the precision measures R2 and MSE are used. In our case; the R2 is almost one and MSE is close to zero. This shows a strong correlation between the measured results and the predicted results. (See Tables 1 to 4). Furthermore, for the sake of time synchronization, during response time measurement, we used the time obtained from time command in Linux; as well as real wall clock time. In order to explore more the behavior of ASR and TTS on multicore processor, the CPU utilization (see Figure 8), the normalized relative speed up and the number of threads with respect to enabled cores are also measured.

The relation between results of measured and predicted is observed from analysis of variance regression output that shows a good fit between measured and predicted results. Moreover, the prism value computed at 95% level of confidence is far less than 0.05 which value rejects the standard null hypothesis that says that there is no correlation between studied variables.

To further validate our study; we used the most common performance metric for voice based application. It is the response time that is measured during the experiment. The same metric is used in predicting model. That model is the linear regression; it is common for predicting model. For the sake of variety, 2 types of multicore processors with different numbers of cores are used in the experiments.

To generalize the performance model, the response time as a function of input length, can be predicted for further length than the experiment range, this is shown by the straight line that indicates the linear increase of the response time as input length grows. Therefore, the linearity of the response time with respect to input length is general for any input length. This generalization is applied on multicore processor that does not apply hyperthreading. E.g., Intel-xeon dual quad core. However, the multicore processor with hyperthreading enabled is not generalizable. As we observed when running experiment on Intel core i-7. To extrapolate the response time for further number of cores, we base our analysis on the number of threads. Generally, the speed up of ASR increases of 7 % at each increase of a core, i.e., one thread for scoring and another for garbage collection. For TTS has found to be sequential, it means that there is no expected

speed up for higher number of cores. However, there are some systems 'parts such as java virtual machine and the garbage collector that benefit from cores increase, therefore the total speed up increases.

6. CONCLUSION

This paper presents a performance evaluation of open source speech recognition and text to speech, i.e., Sphinx-4 and FreeTTS-1.2.2 when these are implemented on multicore processors. The hardware used are, Intel core i-7 quad cores, and Intel-xeon dual quadcore.

In order to use the multicore processor economically, ASR and TTS should be investigated and redesigned. Free and open source help in this. The experiment results on Intel core i-7 cannot be generalized because of the hyperthreading which does not add up to the ASR parallel part neither to TTS. In this paper, the generalization can only be done on multicore with no hyperthreading. The measured average response time is studied as a function of the input length and the number of cores. The number of threads, the relative speed up, and CPU utilization are also measured in order to investigate the performance in detail. The results from the performance model when varying the load length reveals that both ASR and TTS have a linearity property of the response time with respect to the input length; this property tells us that it is relatively easy to generalize the performance model to any input length.

The response time from one number of cores to another is described by the changes of the regression equation coefficient, i.e., the slope of the speedup increase. Both ASR and TTS apply garbage collector which is inherently parallel. It counts for almost 2 % of the total work. Moreover, ASR has acoustic scoring part that runs in parallel; the number of threads is always updated dynamically with the number of cores respectively. The speed up is about 7% per one core increase. And this could be used to extrapolate in order to predict for further than experiment range of cores. The speed up increase from the program itself is trivial because the latter is sequential. However, some system's part like OS, java virtual machine thread and garbage collector benefits from the core increase, hence the TTS total response time get a little increase of about 5% at each core increase. In this paper, the generalization is application specific, i.e., Sphinx-4 and FreeTTS - 1.2.2. To open source development community, based on the findings, it is important to always update the source code in order to accommodate current technology.

7. REFERENCES

- [1] J. R. Lewis, P. M. Commarford, and C. Kotan, 'Web-Based Comparison of Two Styles of Auditory Presentation: All TTS versus Rapidly Mixed Tts and Recordings', in *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, 2006, vol. 50, pp. 723–727.
- [2] C. Niyizamwiyitira, L. Lundberg, and M. Svahnberg, 'Evaluation of Voice-driven Web Application Architecture', in *2012 Eighth International Conference on Signal Image Technology and Internet Based Systems (SITIS)*, 2012, pp. 555–562.
- [3] 'FreeTTS 1.2 - A speech synthesizer written entirely in the Java(TM) programming language'. [Online]. Available: <http://freetts.sourceforge.net/docs/index.php>. [Accessed: 10-May-2013].
- [4] 'Sphinx-4 - A speech recognizer written entirely in the Java(TM) programming language'. [Online]. Available: <http://cmusphinx.sourceforge.net/sphinx4/#source>. [Accessed:08-Jul-2013].
- [5] B. Schauer, 'Multicore processors—A necessity', *ProQuest Discovery Guides1–14*, 2008.
- [6] W. Walker, P. Lamere, P. Kwok, B. Raj, R. Singh, E. Gouvea, P. Wolf, and J. Woelfel, 'Sphinx-4: a flexible open source framework for speech recognition', Sun Microsystems, Inc., Mountain View, CA, USA, 2004.
- [7] H.-L. Lou, 'Implementing the Viterbi algorithm', *Signal Processing Magazine, IEEE*, vol. 12, no. 5, pp. 42–52, 1995.
- [8] K. You, J. Chong, Y. Yi, E. Gonina, C. J. Hughes, Y.-K. Chen, W. Sung, and K. Keutzer, 'Parallel scalability in speech recognition', *IEEE Signal Processing Magazine*, vol. 26, no. 6, pp. 124–135, 2009.
- [9] S. Yoshizawa, N. Wada, N. Hayasaka, and Y. Miyana, 'Scalable architecture for word HMM-based speech recognition and VLSI implementation in complete system', *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 53, no. 1, pp. 70–77, 2006.
- [10] X. Chen, C. Xu, R. P. Dick, and Z. M. Mao, 'Performance and power modeling in a multi-programmed multi-core environment', in *Proceedings of the 47th Design Automation Conference*, 2010, pp. 813–818.
- [11] W. Walker, P. Lamere, and P. Kwok, 'FreeTTS: a performance case study', Sun Microsystems, Inc., Mountain View, CA, USA, 2002.
- [12] V. S. Adve, *Analyzing the behavior and performance of parallel programs*, vol. 1201. Citeseer, 1993.
- [13] A. S. Namin, M. Sridharan, and P. Tomar, 'Predicting Multi-Core Performance: A Case Study Using Solaris Containers', in *Proceedings of the 3rd International Workshop on Multicore Software Engineering, ser. IWMSE*, 2010, vol. 10, pp. 18–25.
- [14] P. Lamere, P. Kwok, E. Gouvea, B. Raj, R. Singh, W. Walker, M. Warmuth, and P. Wolf, 'The CMU SPHINX-4 speech recognition system', in *IEEE Intl. Conf. on Acoustics, Speech and Signal Processing (ICASSP 2003)*, Hong Kong, 2003, pp. 2–5.
- [15] 'Sphinx-4 Transcriber Demo'. [Online]. Available: <http://cmusphinx.sourceforge.net/sphinx4/src/apps/edu/cmu/sphinx/demo/transcriber/README.html>. [Accessed: 10-May-2013].
- [16] 'Alice's Adventures in Wonderland (Project Gutenberg)'. [Online]. Available: <http://www.cs.cmu.edu/~rgs/alice-table.html>. [Accessed: 10-May-2013].
- [17] 'Home—Project Kenai'. [Online]. Available: <http://visualvm.java.net/>. [Accessed: 08-Jul-2013].
- [18] J. O. Rawlings, S. G. Pantula, and D. A. Dickey, *Applied regression analysis: a research tool*. Springer, 1998.
- [19] J. Miles and M. Shevlin, *Applying regression and correlation: A guide for students and researchers*. SAGE Publications Limited, 2001.
- [20] M. D. Hill and M. R. Marty, 'Amdahl's law in the multicore era', *Computer*, vol. 41, no. 7, pp. 33–38, 2008.