Portable Parallel Translation Machine for Multi-Dictionary Systems

Moses E. Ekpenyong
Department of Mathematics, Statistics and Computer science, University of Uyo, P.M.B. 1017, Uyo, 520001, Uyo, Nigeria
E-mail: ekpenyong_moses@yahoo.com Tel: (234)8037933961

Abstract
This paper implements an open source multi-dictionary parallel-translation machine using the Python programming language. The implementation parallelizes the translations of English words into three different languages (German, Ibibio and French). The research model has adaptability for n-languages, which could be implemented by adding n-process threads to the current design and building n-dictionaries in a Python compatible database format. To evaluate the performance of this design, a sequential version of the parallel-translation machine was also developed and compared with the parallel version. It was observed that on the average, for ten (10) existing words randomly selected from a database of four hundred and six (406) words, the processing time for translations in the parallel-translation machine was faster than its sequential version.

Keywords: *DBMS, MT, Python MPI, Simulation, Multi-threading, String Processing*

Introduction

Dictionaries and word translation models are used by various systems, most often in machine translation (MT). In the realm of database management systems (DBMSs), the term concurrency/parallelism is used to denote the ability of more than one database application (or process) to run at the same time. DBMS can allow multiple users access to data at the same time while maintaining integrity and consistency of data. Because DBMSs has the ability to share data among multiple users and multiple applications, the database system should provide a means for managing concurrent access to data thereby ensuring that the data will be maintained in a consistent state, and that the integrity of data will be preserved.

One of the methods of accomplishing this is to enforce an exclusive serial mode of processing database requests. That is, each transaction waits until another transaction has completed its work. However, this type of processing results in performance levels that are simply unacceptable for today's online systems and customer's expectations. As an alternative, the DBMS can manage the access to data through the means of locks which are software mechanisms used in order to allow as much throughput (by maximizing concurrent access to data) as possible while maintaining the integrity and consistency of data.

This paper implements a parallel-translation model that processes multi-databases of translations simultaneously and extracts translations of English words entered by a user. At the end of this paper, a more cost effective tool for portable parallelism that could be modified for other implementations will be accomplished.

Parallel Dictionaries: Related Literature

McEwan, Ounis & Ruthven (2002) describe a system for automatically constructing bilingual dictionary for Cross-Language Information Retrieval (CLIR) applications. They describe how parallel documents can be automatically accessed, filtered and processed to create parallel sentences.

Parallel texts have been applied in several studies on CLIR (Brown, 1998; Davies & Ogden, 1997; Littman, Dumais & Landauer, 1998; Yang, Carbonell, Brown & Frederking, 1998). In Littman et.al. (1998), the latent semantic reduction indexing approach has been applied to a relatively parallel text collection in English with French, Spanish, Greek and Japanese. This approach tries to overcome the problems of lexical matching by using statistically derived conceptual indices instead of individual words for retrieval. It assumes that there exist some underlying or latent structures in word usage that is particularly obscured by variability in word choice. The effectiveness of this approach has not been tested on a large data collection. In Yang et. al. (1998), a corpus-based bilingual term-substitution thesaurus, has been constructed from parallel texts using co-occurrence information. Davies & Ogden (1997) integrate traditional, glossary-based machine transaction technology with Information Retrieval (IR) approaches into Spanish/English CLIR system. These approaches use domain-specific parallel collections, which are costly to obtain.

Recently, there have been several attempts to collect cheap parallel texts from the web. Resnik (1998) and Resnik (1999) were among the foremost researchers to investigate methods to collect parallel/translate texts from the web. Automatic construction of thesauri using statistical techniques is a most common IR technique, (Chen, 2000; Chen and Nie, 2000; Van Rijsbergen, 1999).

Dictionary Building

A lot of work has been done on dictionary building, with various techniques. One good overview is Melamed (2000). There is also an active research area focusing on multi-source translation (for instance, Och and Ney, 2001). Mc Ewan, et. al. (2002) divide the dictionary building stage into three steps: building a matrix of words, normalizing the raw co-occurrence scores in the matrix and making a dictionary listing of terms with the highest co-occurrence probability for each equivalent language term.

Sabot (1986) describes the problem of parallel dictionary lookup as, “given both a dictionary and a text consisting of thousands of words, how can the appropriate definitions be distributed to the words in the text as rapidly as possible?” He discovered and described a parallel dictionary lookup algorithm that makes efficient use of the connection machine (CM) – a bit serial hardware. It is very clear that most natural languages processing applications require an efficient lookup algorithm. Indexing and searching of databases consisting of unformatted natural language text is one of such applications.

Accessing Dictionaries

A dictionary may be defined as a mapping that receives a certain word and returns a set of status bits. Status bits indicate which group of words a certain word belongs to. Some sets that are useful in natural language processing include syntactic categories such as verbs, nouns and prepositions. Programs can also implement semantic information characterization by capturing the structure in an information schema, which specifies the semantic classes of the information domain, the words and phrases that belongs to these classes, and the predicate argument relationships among members of these classes, which are meaningful to the domain.

Sabot (1986) implement dictionary access where a lookup definition of word consists returning a binary number that contains 1’s only in bit positions that corresponds with the groups to which the word belongs. He improved on dictionary building by introducing two sub-modules (sort and scan). A parallel sort is similar in function to a serial sort. It accepts as arguments a parallel data field and a parallel comparison predicate, and sorts among the selected processors so that the data in each successive (by address) processor increases monotonically. A scan algorithm takes an associated function of two arguments, called F, and quickly applies it to data field values in successive processors of say a, b, c, d, e. The scan algorithm produces output fields in some processors with values: a, F(a,b), F(F(a,b),c), F(F(F(a,b),c),d), etc. the major point is that a scan algorithm can take advantage of the associative law and perform a task in logarithmic time. Thus 16 applications of F are sufficient to scan F across 64,000 processors. Sabot (1986) shows one possible scheme for implementing scan, while the scheme is based on a simple linked structure, scan may also be implemented on binary trees, hyper-cubes and other graph data structures.

Design Method

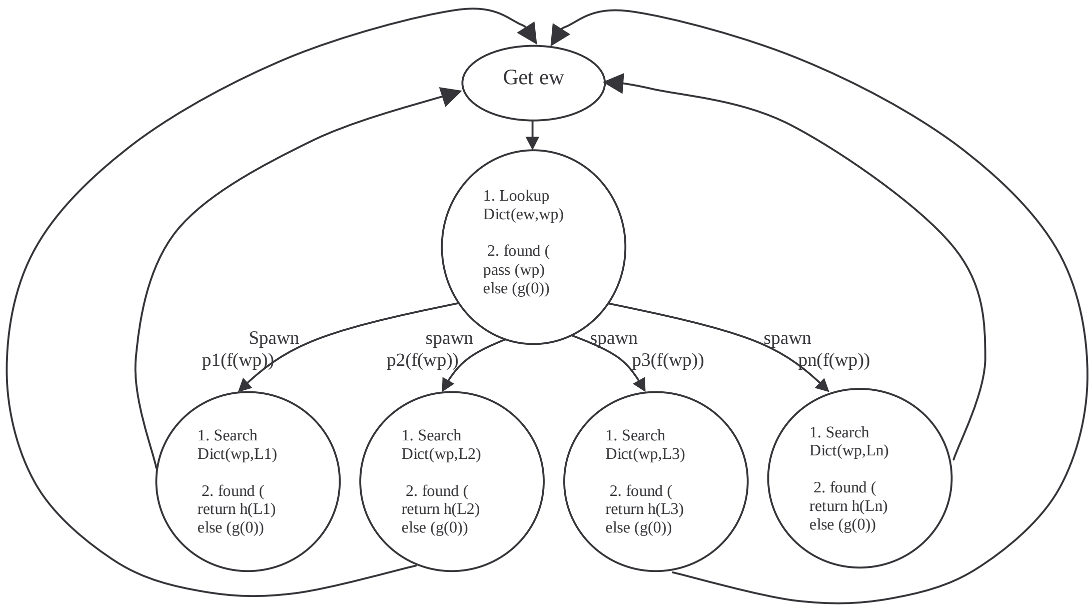
In this section we simulate a multi-dictionary (English, Ibibio, German and French) parallel-translation system using Python programming language. The essence of using Python is to make the program much easier to port, or modify to run on different computers and the Internet. The research implements the multi-program multi-data (MPMD) model. It receives a search word from the operator or user and passes the word position with same search program to the different processors or threads. The dictionaries (databases) are built in Excel (spreadsheet) using accepted Python dictionary data-structure format (i.e. converted to .csv files) to enable processing and updating flexibility. These dictionaries are translations of the English words in the English database. The translations are also directly mapped to the English database, thus eliminating search errors during the parallel search. This implies that all the dictionaries (English, Ibibio, German, and French) contain same number of wordlist. The dictionaries are searched concurrently after spawning the processors. If a search word is successfully matched in a particular dictionary/database, the required translation of that word is returned, else an error message results.

Programming Language Choice

Python has rich string processing functions and its multi-threading facilities are used in the programming to provide the much-needed features of parallelism. Parallelism is emulated in our design using Python's multi-threading facilities. The threads (processes or processors) are simultaneously spawned and run in the random access memory (RAM) of the computer system. The justification of this approach is that the spawned threads can share same memory space. This approach represents a more cost effective solution to parallel computing since it is worthwhile to expand memory than building parallel machines. Python effectively handles parallelism issues such as deadlocks, threading, synchronization and locking.

Research Model

A finite state transducer (FST) that describes the research model is as shown below:


Get ew
1. Lookup Dict(ew,wp) 2. found (pass (wp) else (g(0))
Spawn p1(f(wp))
spawn p2(f(wp))
spawn p3(f(wp))
spawn pn(f(wp))
1. Search Dict(wp,L1) 2. found (return h(L1) else (g(0))
1. Search Dict(wp,L2) 2. found (return h(L2) else (g(0))
1. Search Dict(wp,L3) 2. found (return h(L3) else (g(0))
1. Search Dict(wp,Ln) 2. found (return h(Ln) else (g(0))

Legend: ew - english word, wp - word position, spawn pi(f(wp)) - pass wp function into i process, Search Dict (wp,Li) - search for the ith language equivalent for wp, h(Li) -ith language translation for wp, g(0) - error (word not found in dictionary)

Fig. 1. Multi-lingual Parallel-Dictionary Model

Program Modules
Below shows the Python program modules that implement the FST in fig. 1.:

```
#MULTI-THREADED VERSION
#import python essential files for multi-threading,
etc.
from threading import Thread
import string
import time
import os

#create a thread class for Process one (Ibibio
process)
class thread_ibibio(Thread):

   #define a local constructor for word input and
dictionary file variables
   def __init__(self, posinput, dictfile):
         Thread.__init__(self)
         self.posinput = posinput
         self.dictfile = dictfile
#run ibibio search thread
def run(self):
         #search and locate the english word
position in Ibibio dictionary and
         #extract equivalent translation

         i=0
         #open ibibio file for read
         print "\nProcessing Start time (ms) = ",
time.time()
         inp = open(self.dictfile,"r")
         for line in inp.readlines():
            i=i+1
            if self.posinput == i:
               print "Ibibio Translation: ", line
         #close file
         inp.close()


#create a thread class for Process Two (German
process)
class thread_german(Thread):

   #define a local constructor for word position
input and dictionary file variables
   def __init__(self, posinput, dictfile):
         Thread.__init__(self)
         self.posinput = posinput
         self.dictfile = dictfile

   #run German search thread
   def run(self):

         #search and locate the english word
position in German dictionary and
         #extract equivalent translation
         i=0
         #open german file for read
         inp = open(self.dictfile,"r")
         for line in inp.readlines():
            i=i+1
            if self.posinput == i:
               print "German Translation: ", line
         #close file
         inp.close()

#create a thread class for Process Three (French
process)
class thread_french(Thread):

   #define a local constructor for word position
input and dictionary file variables
   def __init__(self, posinput, dictfile):
         Thread.__init__(self)
         self.posinput = posinput
         self.dictfile= dictfile

   #run French search thread
   def run(self):

         #search and locate the english word
position in French dictionary and
         #extract equivalent translation
         i=0

         #open french file for read
         inp = open(self.dictfile,"r")
         for line in inp.readlines():
            i=i+1
            if self.posinput == i:
               print "French Translation: ", line
         #close file
         inp.close()
```

```
        print "Processing Stop time (ms) = ",
time.time()
        print "\n"
        quit = raw_input("Check another
translation? (y/n): ")
        if quit == "y":
           main()

#Main procedure
def main():

   #request for an english word
   os.system("cls")
   print "\t\tTHREADED EXECUTION OF
MULTI GLOSSARY SYSTEM\n"
   engword=raw_input("Enter an English word
you wish to translate: ")

   #locate the line or word position in english
glossary file
   #declare word position
   wordpos,engwordpos = 0,0
   inp = open("englishglofile.csv","r")
   word=[]
   for line in inp.readlines():
      wordpos = wordpos+1
      word=line
      word1 = word[:-1]

      #if english word is in english glossary file
      if engword == word1:
         engwordpos=wordpos
         break

   #close file
   inp.close()

   if engwordpos == 0:
      print "\nWord not existing in English
Glossary file"
   else:
      #pass word line into Prosesses One, Two and
Three
      processOne =
thread_ibibio(engwordpos,"ibibioglofile.csv")
      processTwo =
thread_german(engwordpos,"germanglofile.csv")
      processThree =
thread_french(engwordpos,"frenchglofile.csv")

      #Start the three processes simultaneously
      processOne.start()
      processTwo.start()
      processThree.start()

main()
```

Program modules listing for the multi-threaded version

```
#SEQUENTIAL VERSION
#import python essential files
import string
import time
import os

#Main procedure
def main():
   #request for an english word
   os.system("cls")
   print "\t\tSEQUENTIAL EXECUTION OF
MULTI GLOSSARY SYSTEM\n"
   engword=raw_input("Enter an English word
you wish to translate: ")

   #locate the line or word position in english
glossary file
   #declare word position
   wordpos,engwordpos = 0,0
   inp = open("englishglofile.csv","r")
   word=[]
   for line in inp.readlines():
      wordpos = wordpos+1
      word=line
      word1=word[:-1]

      #if english word is in english glossary file
      if engword == word1:
         engwordpos=wordpos
         break

   #close file
   inp.close()

   if engwordpos == 0:
      print "\nWord not existing in English
Glossary file\n"
   else:
      print "\nProcessing Start time = ",
time.time()
```

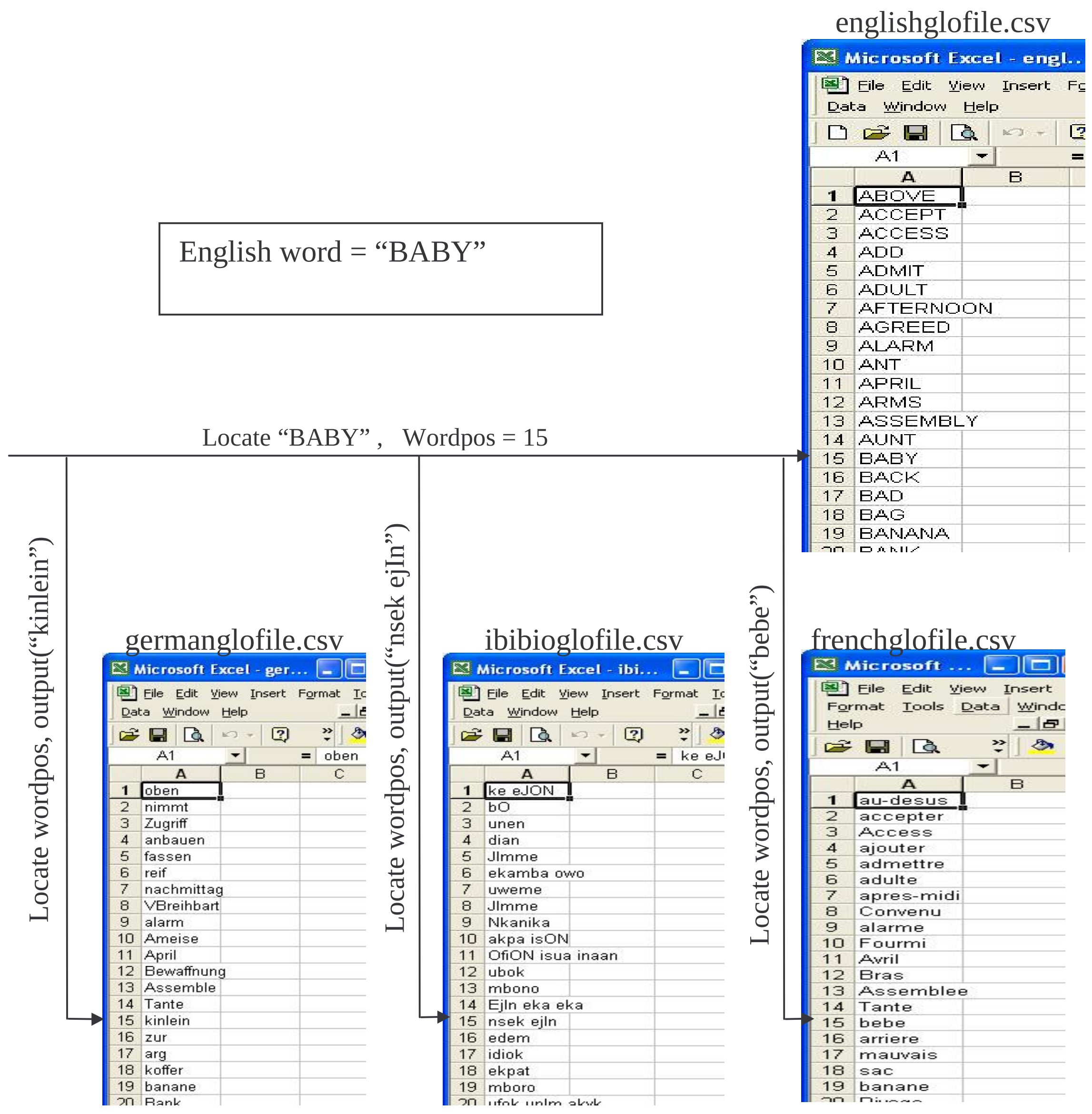
```
    #open ibibio glossary file and search for
word position
    inp = open("ibibioglofile.csv","r")
    i = 0
    for line in inp.readlines():
      i=i+1
      if engwordpos == i:
        print "Ibibio Translation: ", line
    #close file
    inp.close()

    #delay

    #open german glossary file and search for
word position
    inp = open("germanglofile.csv","r")
    i=0
    for line in inp.readlines():
      i=i+1
      if engwordpos == i:
        print "German Translation: ", line
    #close file
    inp.close()

    #open french glossary file and search for
word position
    inp = open("frenchglofile.csv","r")
    i=0
    for line in inp.readlines():
      i=i+1
      if engwordpos == i:
        print "French Translation: ", line
    #close file
    inp.close()

    #delay

    print "Processing Stop time ", time.time()
    print "\n"

    quit = raw_input("Check another
translation? (y/n): ")
    if quit == "y":
       main()

main()
```

Program module listing for the sequential version

The program listing in page 6 is presented to enable readers who are unfamiliar with parallel simulations and may wish to try out the implementation.

Database Implementation

A database illustration of the parallel search implementation is shown below:


englishglofile.csv
English word = "BABY"
Locate "BABY" , Wordpos = 15
Locate wordpos, output("kinlein")
Locate wordpos, output("nsek ejln")
Locate wordpos, output("bebe")
germanglofile.csv
ibibioglofile.csv
frenchglofile.csv
Microsoft Excel - engl..
Microsoft Excel - ger...
Microsoft Excel - ibi...
Microsoft ...
File Edit View Insert Format Tools Data Window Help
A1

englishglofile.csv

	A
1	ABOVE
2	ACCEPT
3	ACCESS
4	ADD
5	ADMIT
6	ADULT
7	AFTERNOON
8	AGREED
9	ALARM
10	ANT
11	APRIL
12	ARMS
13	ASSEMBLY
14	AUNT
15	BABY
16	BACK
17	BAD
18	BAG
19	BANANA

germanglofile.csv

	A
1	oben
2	nimmt
3	Zugriff
4	anbauen
5	fassen
6	reif
7	nachmittag
8	VBreihbart
9	alarm
10	Ameise
11	April
12	Bewaffnung
13	Assemble
14	Tante
15	kinlein
16	zur
17	arg
18	koffer
19	banane

ibibioglofile.csv

	A
1	ke eJON
2	bO
3	unen
4	dian
5	Jlmme
6	ekamba owo
7	uweme
8	Jlmme
9	Nkanika
10	akpa isON
11	OfiON isua inaan
12	ubok
13	mbono
14	Ejln eka eka
15	nsek ejln
16	edem
17	idiok
18	ekpat
19	mboro

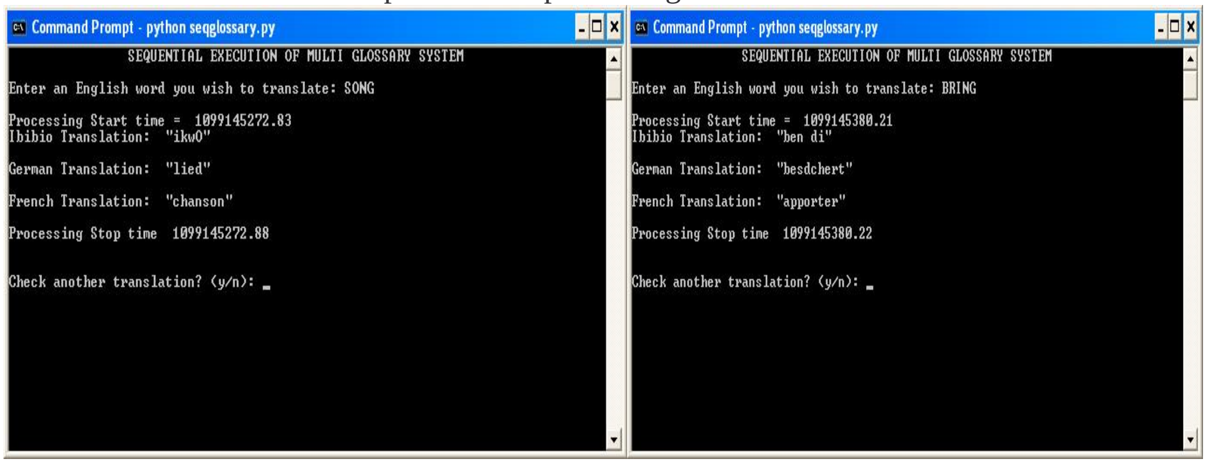
frenchglofile.csv

	A
1	au-desus
2	accepter
3	Access
4	ajouter
5	admettre
6	adulte
7	apres-midi
8	Convenu
9	alarme
10	Fourmi
11	Avril
12	Bras
13	Assemblee
14	Tante
15	bebe
16	arriere
17	mauvais
18	sac
19	banane

Fig. 2. Database implementation of the parallel search.

Sample Input/Output Interfaces

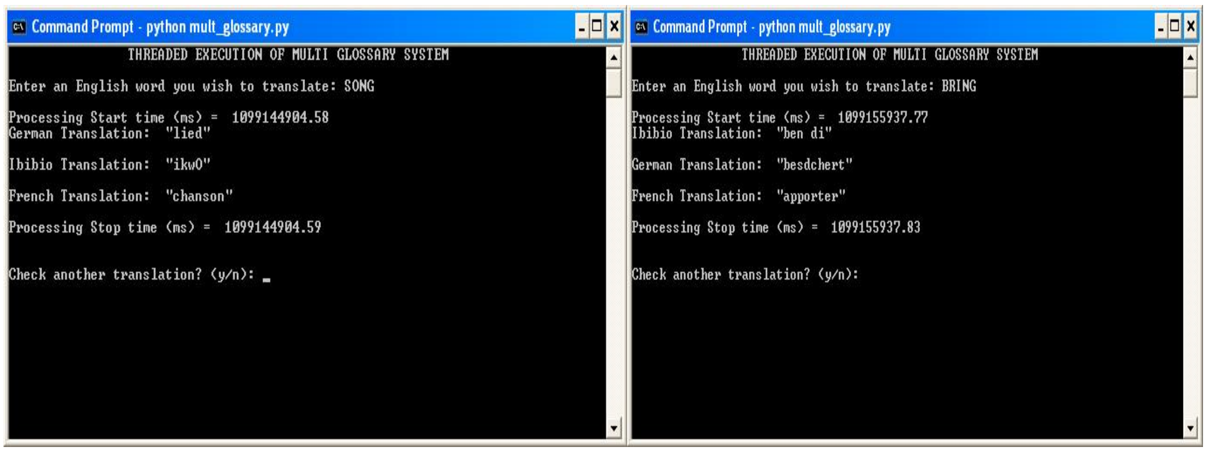
Figs. 3(a) and 3(b) shows the sample input/output interface of the design. The input to the program is an English word that requires translation. The program records using the computer time, the processing start time (time before translation) in milliseconds after accepting an input. It then searches the respective databases for equivalent translations, if a successful translation is matched, the translations are displayed and the time after processing (processing stop time) is recorded. The time difference as recorded in table 1 represents the processing time.



```
Command Prompt - python seqglossary.py
                 SEQUENTIAL EXECUTION OF MULTI GLOSSARY SYSTEM

Enter an English word you wish to translate: SONG

Processing Start time =  1099145272.83
Ibibio Translation:  "ikwO"

German Translation:  "lied"

French Translation:  "chanson"

Processing Stop time  1099145272.88


Check another translation? (y/n): _
```

```
Command Prompt - python seqglossary.py
                 SEQUENTIAL EXECUTION OF MULTI GLOSSARY SYSTEM

Enter an English word you wish to translate: BRING

Processing Start time =  1099145380.21
Ibibio Translation:  "ben di"

German Translation:  "besdchert"

French Translation:  "apporter"

Processing Stop time  1099145380.22


Check another translation? (y/n): _
```

Fig. 3(a). Sequential version input/output interface for the words “SONG” and “BRING”



```
Command Prompt - python mult_glossary.py
                 THREADED EXECUTION OF MULTI GLOSSARY SYSTEM

Enter an English word you wish to translate: SONG

Processing Start time (ms) =  1099144904.58
German Translation:  "lied"

Ibibio Translation:  "ikwO"

French Translation:  "chanson"

Processing Stop time (ms) =  1099144904.59


Check another translation? (y/n): _
```

```
Command Prompt - python mult_glossary.py
                 THREADED EXECUTION OF MULTI GLOSSARY SYSTEM

Enter an English word you wish to translate: BRING

Processing Start time (ms) =  1099155937.77
Ibibio Translation:  "ben di"

German Translation:  "besdchert"

French Translation:  "apporter"

Processing Stop time (ms) =  1099155937.83


Check another translation? (y/n):
```

Fig. 3(b). Parallel version input/output interface for the words “SONG” and “BRING”

Presentation and Discussion of Results

In this section, the results obtained from the research are presented and discussed. The performance of both the sequential and parallel versions of the machine was evaluated by implementing the system with databases of four hundred and six (406) words/translations. The sequential version executes the different translations in a serial order while the parallel version executes the translations simultaneously.

Table 1 shows the performance results for both program versions.

Table 1: Summary Result of ten randomly selected words

S/n	Eng. Word	WP	SEQ_{Transl} time (ms)			$PARLL_{Transl}$ time (ms)		
			Start search	Stop search	T_{diff}	Start search	Stop search	T_{diff}
1	ABOVE	1	1099143562.36	1099143562.37	0.01	1099146362.84	1099146363.81	0.07
2	BRING	50	1099145380.21	1099145380.22	0.01	1099155937.77	1099155937.83	0.06
3	COME	80	1099143598.51	1099143598.53	0.02	1099144616.36	1099144616.38	0.02
4	FORK	140	1099143873.20	1099143873.24	0.04	1099144725.43	1099144725.45	0.02
5	HOSPITAL	174	1099145535.40	1099145535.44	0.04	1099145028.14	1099145028.16	0.02
6	JUMP	184	1099143668.56	1099143668.61	0.05	1099144673.75	1099144673.77	0.02
7	LIFT	204	1099144505.11	1099144505.15	0.04	1099145103.17	1099145103.19	0.02
8	SONG	326	1099145272.83	1099145272.88	0.05	1099144904.58	1099144904.59	0.01
9	WEPT	392	1099144024.71	1099144024.77	0.06	1099144644.24	1099144644.26	0.02
10	ZERO	406	1099145889.19	1099145889.25	0.06	1099146168.18	1099146168.20	0.02

Legend: Eng. word: English word, WP: Word Position in English dictionary/database
SEQ_{Transl} : Sequential version translation time in milliseconds, $PARLL_{Transl}$: Parallelized version translation time in milliseconds, T_{diff}: Time difference (ms) – Processing time.

Speedup is the extent to which more hardware can perform some task in less time than the original system. Speedup is an important property for measuring the performance goals of parallel processing. With good speedup, the system response time could be reduced. The formula below measures the speedup:

$$\text{Speedup} = \frac{\text{Sequential time spent}}{\text{Parallel time spent}}$$

The speedups at different word positions are as shown in Table 2 below:

Table 2. Speedups obtained from the research

Word position	1	50	80	140	174	184	204	326	392	406
Speedup	0.1429	0.1667	1	2	2	2.5	3	5	5	3

From table 1, a plot of the processing time vs word position reveals that the translation time for the sequential version increases as words to be translated approaches the end of file (fig. 4(a).), while for the multi-threaded translation version, the processing time decreases as words to be translated approaches the end of file (fig. 4(b)).

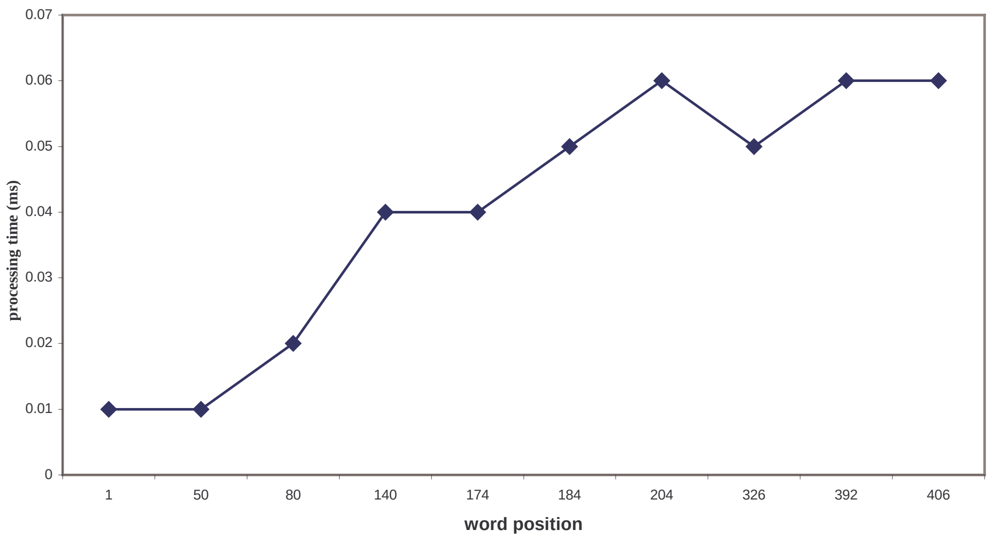


Fig. 4(a). A plot of processing time vs word position for the sequential translation system.

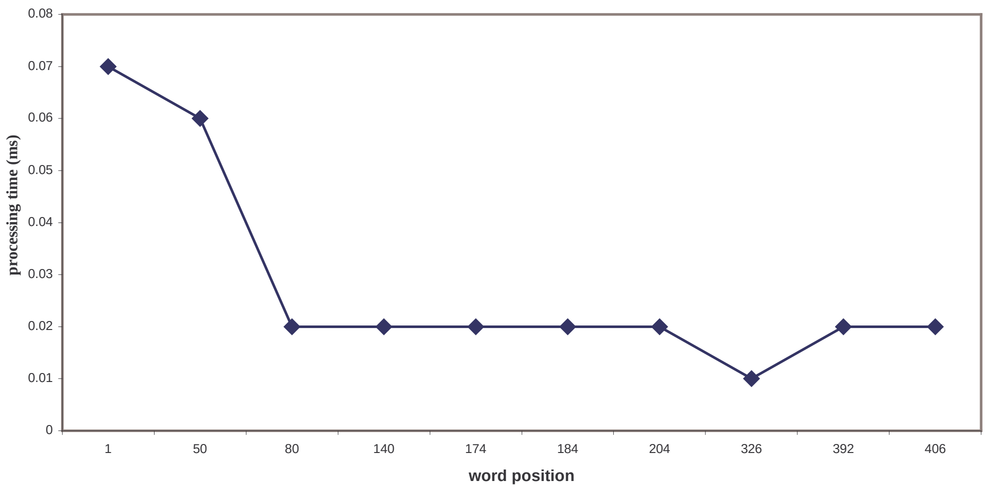


Fig. 4(b).A plot of processing time vs word position for the parallel translation system.

From the above plots we observe that for parallelism to be noticed on microcomputers, the processors must be efficiently utilized, (Ekpenyong & Boukari, 2004; Ekpenyong, Boukari & Edemenang, 2005). Better speedups could be obtained by implementing large databases/dictionaries. This is evident in Table 1, where the speedup appreciates as the processing distance increases. The design is also suitable for implementing complex database systems on microcomputers.

Conclusion

This work has provided an adaptable model/framework for parallel multi-dictionary translation systems. The research has also provided facilities for n-language translations. The framework is most suitable for distributed databases and complex scientific computations/ processing systems. The efficiency of the design implementation can be appreciated with increase in the volume of processed data.

References

1. Brown, R. D. (1998). Automatically Extracted Thesauri for Cross-Language IR: When better is worse, 1st Workshop on Computational Terminology (Computerm): 15-21.
2. Chen, J. (2000). Parallel Text Mining for Cross-Language Information Retrieval using a Statistical Translation Model. M.Sc. Thesis, University of Montreal. http/www.iro.umontreal.ca/~chen/thesis/ model.html.
3. Chen, J. & Nie, J. Y (2000). Parallel Web Text Mining for Cross-Language IR. In Proceedings of RIAO-2000. “Context-Based Multimedia Information Access”, Paris: 12-14.
4. Davies, M. W. & Ogden, W. C. (1997). QUILT: Implementing a Large Scale Cross-Language Text Information Retrieval (ACM SIGIR ’97), Philadelphia: 92-98.
5. Ekpenyong, M. E. & Boukari, S. (2004). Simulating SPMDs on Sequential Machines. Journal of Computer Science & Its Applications. Vol. 4. No. 2: 40-49.
6. Ekpenyong, M. E.; Boukari, S & Edemenang, E, J. (2005). Simulating MPMDs on Sequential Environments. Ultra Scientist of Physical Sciences, Vol. 17, No. 3(M): 477-486.
7. Littman, M. L.; Dumais, S. T. & Landauer, T. K. (1998). Automatic Cross-Language Information Retrieval using Latent Semantic Indexing. In Grefenstette, G. (ed:): Cross-language Information Retrieval, Kluwer Academic Publishers: 51-62.
8. McEwan, C. J.; Ounis, L. & Ruthven, I. (2002). Building Bilingual Dictionaries from Parallel Web Documents, ECIR: 303-323, http://www.citeseer.ist.Psu.edu/655761.html.
9. Melamed, I, D. (2000). Models of Translational Equivalence among Words. Computational Linguistics, 26: 221-249.
10. Och, F. J. & Ney, H. (2001). Statistical Multi-Source Translation. In Proceedings of MT Summit VIII: 253-258.
11. Resnik, P. (1998). Parallel Strands. A Preliminary Investigation into Mining the Web for Bilingual Text. In Proceedings of the AMTA - 98 Conference.
12. Resnik, P. (1999). Mining the Web for Bilingual Text. In Proceeding of the International Conference of the Association of Computational Linguistics (ACL-99), College Park, Maryland.
13. Sabot, G. W. (1986). Bulk Processing of Text on a Massively Parallel Computer. In proceedings of 24th Annual Meeting on Association for Computational Linguistics, 128-135.
14. Van Rijsbergen, C. J. (1999). Information Retrieval. 2nd Edition. CD-ROM Version. http://www.dcs.gla.ac.uk/Keith/Preface.html.
15. Yang, Y.; Carbonell, J. G.; Brown, R. D. & Frederking, R. E. (1998). Translingual Information Retrieval: learning from bilingual Gespora, Artificial Intelligence, 103:323-345.

Article received: 2006-10-25