Optimising EM

James Christian Read

September 4, 2009

MSc in High Performance Computing
The University of Edinburgh
Year of Presentation: 2009
Abstract

The expectation maximisation algorithm is frequently used for the unsupervised learning of bilingual translation dictionaries for statistical machine translation systems. A popular implementation, Giza++, is notoriously slow and the source of long waits for many researchers in the field of statistical machine translation. Attempts to parallelise Giza++ have had some success but no consideration has been made of serial optimisations.

In this dissertation the use of several different data structures is considered and their effects on the performance of EM for this task are explored. A naïve matrix, a dense matrix, a hash table and a hybrid hash table with a dense matrix software cache table implementation were all experimented with. The matrix versions are not able to process all sentences in a data set because of their wasteful use of memory. However, a naïve matrix is capable of processing a surprisingly large percentage of sentences. A dense matrix is capable of processing a higher percentage with the same memory constraints. The matrix data structures were by far the fastest.

A hash table is capable of storing more information in memory and is less wasteful of memory while still being reasonably fast to process. With a hash table it was shown to be possible to process all the sentences of a typical corpus on a single node of HECToR but with the slowest performance. This performance was improved upon by the use of hybrid data structures which attempt to exploit the hash table’s conservative memory usage and the dense matrix’s high performance by using a dense matrix as a software cache to supplement the hash table. A simple heuristic for automatically selecting the optimal size of software cache table remains elusive.
Contents

1 Introduction 1

2 Background theory 2
  2.1 Background Theory Introduction . . . . . . . . . . . . . . . . . . . . . . . . 2
  2.2 Expectation Maximisation Algorithm . . . . . . . . . . . . . . . . . . . . . 2
    2.2.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
    2.2.2 EM Explained by Example - Word Alignment Task . . . . . . . . . . 3
    2.2.3 Formal Definition of EM . . . . . . . . . . . . . . . . . . . . . . . 4
  2.3 Serial Expectation Maximisation . . . . . . . . . . . . . . . . . . . . . . . . 6
    2.3.1 Word Alignment . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
    2.3.2 IBM Model 1 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
    2.3.3 Giza and Giza++ . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
  2.4 Parallel Expectation Maximisation . . . . . . . . . . . . . . . . . . . . . . . . 9
    2.4.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
    2.4.2 MGiza++ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
    2.4.3 PGiza++ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
    2.4.4 MapReduce . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
    2.4.5 Communication Patterns . . . . . . . . . . . . . . . . . . . . . . 14
  2.5 Background Theory Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . 14

3 Optimising EM - IBM Model 1 15
  3.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
  3.2 Computational Cost of EM - IBM Model 1 . . . . . . . . . . . . . . . . . . . . 15
    3.2.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
    3.2.2 Cost of E-Step . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
    3.2.3 Cost of M-Step . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
    3.2.4 Total Cost of EM IBM Model 1 . . . . . . . . . . . . . . . . . . . . 17
    3.2.5 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
  3.3 Data Preparation and Data Structures . . . . . . . . . . . . . . . . . . . . . . 18
    3.3.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
    3.3.2 The Dictionaries . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
    3.3.3 Digital Sentences . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
  3.4 EM and The Translation Table (Counts, Totals and Probabilities) . . . . 29
    3.4.1 2D Array - Matrix . . . . . . . . . . . . . . . . . . . . . . . . . . 29
    3.4.2 Hash Table . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4.3 Hybrid - Hash Table with Cache Table</td>
<td>40</td>
</tr>
<tr>
<td>3.4.4 Summary</td>
<td>41</td>
</tr>
<tr>
<td>4 Parallel EM</td>
<td>43</td>
</tr>
<tr>
<td>4.1 Shared Memory EM IBM Model 1</td>
<td>43</td>
</tr>
<tr>
<td>4.2 Distributed EM IBM Model 1</td>
<td>44</td>
</tr>
<tr>
<td>5 Conclusions</td>
<td>47</td>
</tr>
</tbody>
</table>
# List of Tables

3.1 Statistics showing workload of E-Step and M-Step from the French-English Europarl corpus data ........................................ 18
3.2 This shows the possible ranges of unsigned values when using datatypes of different length to store them. ............................. 20
3.3 Results of word and sentence counting experiments with Europarl data. ................................................................. 25
3.4 Memory consumption and time to build of dictionaries for Europarl corpus data. ......................................................... 26
3.5 Naive matrix performance. ................................................... 32
3.6 Dense matrix performance. .................................................. 34
3.7 Hash table performance. ..................................................... 39
3.8 Hybrid hash table with cache table performance. ................. 40
3.9 Hybrid hash table with cache table performance. ................. 41
3.10 Hybrid hash table with cache table performance. ............... 42
4.1 Shared memory timings with synchronisation disabled on dense matrix version. ......................................................... 43
4.2 MPI timings of E-Step and C-Step using dense matrix version on English-French Europarl corpus. ................................. 45
List of Figures

3.1 Naive distribution of co-occurrences. y axis = source words, y axis = target words ........................................................... 33
3.2 Condensed distribution of co-occurrences. y axis = source words, y axis = target words ......................................................... 35
Acknowledgements

This project would not have been possible without the continued guidance and support of its supervisors Adam Carter and Iain Bethune. Also, David Henty was instrumentally helpful with how to binary search the amount of available memory. Mark Bull was a great source of help with understanding issues pertaining to shared memory parallelisation and understanding the effects of serial optimisations.

Philipp Koehn and Miles Osborne were of great help in explaining the finer points of the EM algorithm and the internals of Moses and of GIZA++.

Jason Wolfe and Qin Gao also helped greatly in explaining their parallelisations of EM IBM Model1.
Chapter 1

Introduction

This dissertation summarises experimentation with the optimisation of an implementation of the expectation maximisation algorithm known as IBM Model 1 which is used for the unsupervised learning of bilingual translation dictionaries for statistical machine translation systems.

In Chapter 2 the relevant background literature is reviewed which explains EM, explains its application to IBM Model 1 and the most relevant applications of IBM Model 1 and its parallelisation.

In Chapter 3 experimentation with using different data types for executing IBM Model 1 is detailed.

In Chapter 4 a template parallelisation scheme for EM is presented with focus on shared memory, distributed message passing and hybrid architectures.

In Chapter 5 conclusions are presented.
Chapter 2

Background theory

2.1 Background Theory Introduction

In this chapter the EM algorithm is introduced and explained with reference to the example of the unsupervised learning of translation dictionaries for statistical machine translation systems. A popular implementation of EM, GIZA++, is introduced and its importance to the machine translation research community is highlighted.

Two attempts to parallelise GIZA++, Pgiza++ and Mgiza++, are introduced and summarised. The prospect of using Google’s MapReduce for EM is reviewed and finally, the work of Jason Wolfe on alternative communication schemes for parallel EM is summarised.

2.2 Expectation Maximisation Algorithm

2.2.1 Introduction

The EM algorithm (expectation maximisation) is a statistical learning algorithm. The algorithm is broadly applicable and has been used for unsupervised learning tasks in many completely unrelated fields. While the algorithm, in various forms, had already been used for some time by various statistical researchers it is generally recognised that the paper that formalised and consolidated our understanding of the EM algorithm is the Dempster, Laird and Rubin paper of 1977 published in the Journal of the Royal Statistical Society [1].

The precise mathematical definitions of the EM algorithm presented by Dempster, Laird and Rubin [1] are not easy reading and may, perhaps, be even more difficult to understand for readers without a substantial statistical maths background. Let us first, therefore, consider perhaps the simplest explanation of the EM algorithm with reference to
a common learning task that EM is frequently used for. Perhaps the most popular and well known use of the EM algorithm presently is that of the unsupervised learning of a TM (translation model) from a bilingual corpus such as the Europarl corpus [2] or the Canadian Hansards [12] - see Koehn’s online lecture notes [4] for the source of the following explanation.

### 2.2.2 EM Explained by Example - Word Alignment Task

A TM is basically a table of database like entries (rows of related fields). Each row in the database represents a source word and a target word that are translations of each other and finally a probability score which shows the relative frequency that such a translation was encountered in a training data set. A few typical rows in an English to French TM might look like the following:

<table>
<thead>
<tr>
<th>Source Word</th>
<th>Target Word</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>the</td>
<td>le</td>
<td>0.4</td>
</tr>
<tr>
<td>the</td>
<td>la</td>
<td>0.3</td>
</tr>
<tr>
<td>the</td>
<td>l’</td>
<td>0.2</td>
</tr>
</tbody>
</table>

As can be seen by simple inspection of the TM there are two basic features of the translation model:

1. The alignments i.e. which target word translates which source word
2. The parameters i.e. the relative frequency that such an alignment is observed

Both of these unknowns need to be learned in order to construct a reliable TM. The implications for the ML (machine learning) task of the unsupervised learning of a TM is that this is a chicken and egg problem. That is to say that there are two unknowns which are dependent upon each other. If we already knew the alignments we could directly observe the parameters by simply inspecting the bilingual corpus and counting the number of times each alignment occurs. If we already knew the probabilities of alignment but not the alignments we could directly observe the alignments by inspecting the corpus and finding the alignments which fit the probabilities. The EM algorithm is broadly applicable to problems of this type - that is to say, problems with two dependent unknowns. This is how it basically works in the case of word alignment:

1. Make an initial guess of alignments and parameters e.g. each source word is equally likely to align with each target word
2. Make a guess at the alignments based on the probabilities learned in step 3
3. Make a guess at the parameters based on the alignments learned in step 2
4. Iterate steps 2 and 3 until convergence

In a nutshell, the above procedure is EM. For more details and illustrative slides see Koehn’s original lecture notes [4] - the inspiration behind this explanation. Depending
on the starting point given in step 1, the initialisation stage, EM will converge on one local maximum. There may be many different local maxima and EM is not guaranteed to converge on the best local maximum (i.e. The global maximum). Convergence on which local maximum depends on the starting point provided.

2.2.3 Formal Definition of EM

The EM algorithm owes its name to Dempster, Laird and Rubin [1] who coined the name in these words: Since each iteration consists of an expectation step is followed by a maximisation step we call it the EM algorithm. In general terms they define the algorithm as presents a general approach to iterative computation of maximum-likelihood estimates when the observations can be viewed as incomplete data.

They provide a mathematical description of what they mean exactly by incomplete data:

The term 'incomplete data’ in its general form implies the existence of two sample spaces Y and X and a many-one mapping from X to Y. The observed data y are a realisation from Y. The corresponding x in X is not observed directly, but only indirectly through y.

Perhaps a reference back to our word alignment example may alleviate some of the pain of trying to decipher what this actually means. In general terms we are looking for complete data i.e. x e.g. the complete TM - both alignments and associated probabilities. But what we actually have is incomplete data i.e. y e.g. an exhaustive list of co-occurring source word - target word pairs but no explicit knowledge of which ones are mere coincidence and which ones are genuinely translations of each other or how often. A human with enough time and patience could look at y and by process of experimentation and induction eventually understand which co-occurrences are genuine translations and how often. That is to say that the TM can be indirectly observed from the exhaustive list of co-occurrence pairs i.e. the complete data can be indirectly observed from the incomplete data - x can be indirectly observed from y.

This implies the existence of a function which maps x not directly to y but indirectly. In fact Dempster, Laird and Rubin go on to state that More specifically, we assume there is a mapping x-y(x) from X to Y, and that x is known only to lie in X(y), the subset of X determined by the equation y=y(x), where y is the observed data. We refer to x as the complete data even though in certain examples x includes what are traditionally called parameters. This should now start to become clear with reference to our word alignment example. The co-occurrences are the incomplete but directly observable data y. The alignments are the indirectly observable yet highly desirable complete data x. However, there are often parameters associated with x. And, in fact, that is exactly what we observe with the word alignment problem. Each word can align to many other words and it is necessary to have the probabilities of each alignment in order to have a complete understanding of x - the complete data.

Dempster, Laird and Rubin go on to provide mathematical reasoning for definitions
of EM both for the special case of regular exponential-family forms and for the general case. The mathematical reasoning and proofs that lead to these definitions is beyond the scope of this dissertation but the interested reader is urged (with the aid of an aged and experienced statistician) to read, decipher and understand section 2 of Dempster, Laird and Rubin [1] to gain a well founded understanding of EM. In short, their mathematical formulas translate into the following two step process:

1. E-Step: Estimate the complete data sufficient statistics

2. M-Step: Choose values for the parameters which maximise the estimate

These two steps correspond to that suggested in our word alignment example. That is to say we guess the alignments by estimating the complete data sufficient statistics. And then we choose values for the probabilities of each alignment which maximise the alignments estimated. Dempster, Laird and Rubin show that iterating over this two step process will eventually converge on a single local maximum (one of many possible local maxima) which is dictated by the initialisation of the model (the complete data with parameters).

Dempster, Laird and Rubin show by many examples in section 4 [1] that the EM algorithm is broadly applicable to many generic problem spaces such as:

- Missing Data
- Grouping, Censoring and Truncation
- Finite Mixtures
- Variance Components
- Hyperparameter Estimation Iteratively Reweighted Least Squares

In fact, they also show by means of many references which the interested reader is invited to follow that the EM algorithm had been used in many different seemingly unrelated fields by many researchers in different forms but with the same basic high level algorithm - the EM algorithm. To this end, when this paper was read before the Royal Statistical Society in 1976 it was immediately received with 2 accepted votes of thanks. Since then it has become firmly established in the literature as the authoritative paper on EM and EM has become an essential tool and well known algorithm in every serious machine learning researcher’s toolbox. It is, therefore, completely in place that Dr J. A. Nelder gave his vote of thanks with these opening words:

*There are times when it seems that our subject is becoming more and more fragmented with people quarrying vigorously in smaller and smaller corners and becoming more and more isolated from each other in consequence. It is a particular pleasure, therefore, to be able to speak to a paper which seeks, and achieves, a synthesis by exhibiting diverse procedures as special cases of a general one.*

We concur with the sentiments of Nelder and the benefits of a complete understanding of the optimisation issues which pertain to the EM class of algorithms should be ob-
vious to any who have eagerly waited hours for EM to complete on a data set of any considerable size.

### 2.3 Serial Expectation Maximisation

#### 2.3.1 Word Alignment

**Introduction**

In the previous section we explained the EM algorithm with constant reference to the unsupervised learning of a TM via word alignment of source words with target words. This is a fertile area of research for the following reasons. MT (machine translation) is the task of automatically translating text from one language into a corresponding text in another language. The dream of being able to perform FAHQT (fully automated high quality translation) is as old as the conception of the computer itself and quite literally billions have been spent in this area with limited results. In fact, in 1966 most MT funding dried up based on the condemning quality of the ALPAC report [5] which concluded that it is cheaper to use human translators than to develop and maintain MT systems. For several decades researchers interested in MT had to get funding under the pretence of researching a related NLP (natural language processing) field the expense of which could be justified with other means.

Up until 1993 rule based mechanisms had dominated MT research. But observing that NLP and speech processing fields had already benefited from the use of statistical methods Brown et al. shook the MT research community with their presentation of CANDIDE [6] and funding sources for MT began to open up again. IBM’s CANDIDE system gave birth to the field of SMT (statistical machine translation) which has caught the attention of big players like Google Translate and the EU. This has led to the availability of the Europarl corpus [2] via high calibre researchers like Koehn. It has also led to the availability of literally millions of euros of MT research funds for the EuroMatrix project [13].

**IBM Models 1 - 5**

The CANDIDE system attacks the problem of machine translation statistically. The details of the method are beyond the scope of this dissertation but an essential component of the attack is a well trained TM. In general, the better trained the TM the better results an SMT system can produce. In a 1993 paper Brown et al. [7] described 5 implementations of EM algorithms which they used to train a TM using French-English data from the Canadian Hansards. These five models are referred to in the literature since then as IBM Models 1 - 5.
IBM Model 1 is a typical EM implementation. In the initialisation stage it is assumed that each word has an equal probability of aligning with any other word in the corpus. No tricks are used in the initialisation stage. Output from IBM Model 1 is used as the input for IBM Model 2. IBM Model 2 makes the assumption that position in the sentence affects the probability of alignment. This is probably more true for languages with a similar word order. Output from IBM Model 2 is the input for IBM Model 3. IBM Model 3 differs from IBM Model 2 in that the number of target words that align to a source word is determined before determining what they are. The output of IBM Model 3 is given as input to IBM Model 4. For IBM Model 4 the target words which are aligned to a source word in any sentence are influenced by which other target words will be aligned to the source word. IBM Models 3 and 4 are deficient in the sense that they waste some of their probability space. IBM Model 5 is very much like IBM Model 4 except that it is not deficient.

All 5 of these models make use of an implementation of the EM algorithm. The most fundamental stage is IBM Model 1, which must always run, as the input to subsequent models. An investigation of the parallelisation of IBM Model 1 should reveal insights which are portable to the parallelisation of all of the Models and EM algorithms in general.

### 2.3.2 IBM Model 1

The maths of IBM Model 1 is formally described in [7]. For our purposes it will be much more useful to take a quick look at the pseudocode from [4]

```plaintext
initialize t(e|f) uniformly
do until convergence
    set count(e|f) to 0 for all e,f
    set total(f) to 0 for all f
    for all sentence pairs (e_s,f_s)
        for all words e in e_s
            total_s(e) = 0
        for all words f in f_s
            total_s(e) += t(e|f)
        for all words e in e_s
            for all words f in f_s
                count(e|f) += t(e|f) / total_s(e)
                total(f) += t(e|f) / total_s(e)
        for all f
            for all e
                t(e|f) = count(e|f) / total(f)
```

This code clearly shows an initialisation step:
initialize \( t(e|f) \) uniformly

Looping until convergence:

do until convergence

An E-Step:

\[
\text{set count}(e|f) \text{ to 0 for all } e,f \\
\text{set total}(f) \text{ to 0 for all } f \\
\text{for all sentence pairs } (e_s,f_s) \\
\text{for all words } e \text{ in } e_s \\
\quad \text{total}_s(e) = 0 \\
\quad \text{for all words } f \text{ in } f_s \\
\quad \quad \text{total}_s(e) += t(e|f) \\
\quad \text{for all words } e \text{ in } e_s \\
\quad \quad \text{for all words } f \text{ in } f_s \\
\quad \quad \quad \text{count}(e|f) += \frac{t(e|f)}{\text{total}_s(e)} \\
\quad \quad \quad \text{total}(f) += \frac{t(e|f)}{\text{total}_s(e)}
\]

And an M-Step

\[
\text{for all } f \\
\quad \text{for all } e \\
\quad \quad t(e|f) = \text{count}(e|f) / \text{total}(f)
\]

2.3.3 Giza and Giza++

As a result of renewed interest in MT caused by the innovation of SMT pioneered by Brown’s IBM team funding became available again for MT research at academic institutions. By the end of the nineties Kevin Knight had a strong research team making leaping advances in SMT and tweaking the IBM models. This led to the Egypt SMT toolkit [14]. Amongst other tools Giza was a part of the Egypt toolkit. Giza used implementations of IBM Models 1, 2 and 3 for unsupervised word alignment. Giza++ features extensions to Giza including implementations of IBM Models 4 and 5 and HMM alignment as an optional replacement for IBM Model 2.

Giza and Giza++ are written in C++ and make extensive use of the STL library. The extensions were largely designed and written by Franz Joseph Och who now leads the MT research team at Google [20]. Och has contributed richly to the literature of SMT
and unsupervised learning of a TM from bilingual corpora. He is largely responsible for the move from using IBM Model 2 to HMM alignment [8].

Giza++ is extremely successful and has gained popularity in SMT research. After gaining his PhD which partly involved implementing Pharaoh [15], Koehn became a leader in SMT research and was quickly taken into Edinburgh University’s ICCS (Institute for Communicating and Collaborative Systems) research team [16]. Koehn and his team developed and maintain Moses [17] the leading open source SMT system that is used by SMT researchers across the globe. The Moses system relies on and is heavily integrated with Giza++ for the unsupervised training of TM’s.

Online at [18] is a breakdown of the time it takes to train Moses to be ready to perform translation of unseen text for a given language pair. This involves the training of an LM (language model), a TM and the extraction and scoring of phrases amongst other tasks. On 15/08/2009 it is reported that training of a TM from a 16 million German-English Europarl corpus takes about 16 hours - the lion’s share of the preparation time to get Moses up and running. This training must be performed every time a better data set is made available or every time a bug fix in the Moses training scripts can lead to a better TM. Also, the improvement of TM alignment is an active field of research and many researchers have to wait such lengthy periods of time to view their results of any tweaks experimented with. The optimisation of the EM algorithms which power Giza++ would clearly be of immense time saving benefit to many researchers across the globe.

2.4 Parallel Expectation Maximisation

2.4.1 Introduction

As we have already seen the pseudocode for IBM Model 1 can be split nicely into an E-Step and M-Step. The E-Step updates global counts and totals for each and every co-occurrence in the bilingual corpus. It does this by looping over sentences and then nested looping over each source word and target word in each sentence. As the update of global counts and totals is a cumulative incremental operation it is possible to follow one of two parallelisation strategies:

- Increment the global counts and totals in synchronised shared data structures
- Increment local counts and totals and then reduce the sum of the local counts and totals into the global counts and totals

The M-Step is much more straightforward to parallelise, in theory, as there is no need to use synchronisation or local specific copies of data structures. The M-Step merely updates each alignment-parameter pair i.e. the probability of each alignment. Parallelising the M-Step is merely a case of dividing the work amongst processes/threads (at least theoretically speaking).
2.4.2 MGiza++

Introduction to MGiza++

MGiza++ is an attempt to parallelise Giza++ for shared memory architectures by Qin Gao [9]. Gao parallelises for HMM and IBM Models 1, 3 and 4 however, we shall concentrate on his parallelisation strategy for Model 1 here. MGiza++ uses POSIX threads and achieves parallelisation of the E-Step of IBM Model 1 by updating synchronised shared data structures of counts and totals.

Parallelisation of MGiza++

In Giza++ the counts are stored in the TTable data structure which starts life as a data base of counts for every co-occurrence pair in the bilingual corpus but is pruned of co-occurrences with near zero values regularly. Despite these attempts to keep the size of the TTable down Gao rightly observes that it is impractical to store local copies of the TTable in a shared memory architecture. For this reason Gao wisely adopts the strategy of updating the counts in the TTable in synchronised fashion. He also rightly updates the totals in a similar synchronised fashion. To avoid collisions when updating these data structures Gao reports that he locks on a foreign word so that updates cannot be made to counts or totals for the same foreign word at the same time by more than one thread. Load balancing is achieved by threads polling for a new sentence each time they have finished with the present one.

Gao tested his solution on 1, 2 and 4 processors for an alignment task of English-Spanish Europarl data containing 900 thousand sentence pairs, 20 million English words and 20 million Spanish words. He observed that for one thread the completion time of Model 1 did not deviate from that of Giza++. For two processors the speed up was less than linear. Gao obtained a speedup of 1.61 (from 2,167 seconds to 1,352 seconds for an English-Spanish data set of 20 million words for each language and 900,000 sentence pairs). For four processors he obtained a speedup of 2.33 (from 2,167 seconds to 928 seconds for the same data set).

Critique of MGiza++

While MGiza++ was definitely a step in a much needed direction there are many criticisms the work can come under:

- No discussion of the possibility of serial optimisation of Giza++ or of the EM algorithm. It is assumed that Giza++ is already running as fast as it can in serial.
- As MGiza++ is not implemented with OpenMP the user is required to supply the program with the number of processors it will run on rather than the program figuring this out dynamically.
An OpenMP implementation could prove to be more easily maintainable.

The use of the TTable data structure is much slower to lookup than a direct addressing scheme. There is no discussion of this in Gao’s paper.

Gao’s algorithm also probably spends a lot of time requesting new sentences because of its simplistic load balancing scheme. Perhaps an affinity load balancing scheme could have reduced this overhead.

There is also no discussion of parallelising the M-Step in MGiza++. This could be done with no synchronisation overhead in embarrassingly parallel fashion.

Finally, there is mention of use of I/O in Gao’s paper to output alignments into files. There is clearly no need for any writes to disk between models and everything that can be done in memory should be done in memory if performance is the desired outcome.

In summary, MGiza++ is a welcomed step in the right direction but there is no discussion of the issues of optimising a serial version of the IBM Model 1 EM implementation.

2.4.3 PGiza++

Introduction to PGiza++

PGiza++ is another attempt to parallelise Giza++ by Qin Gao [9]. This time for distributed architectures. Again, Gao parallelised HMM and IBM Models 1, 3 and 4 but we will again stay focused on IBM Model 1. Basically speaking, PGiza++ parallelises the E-Step of IBM Model 1 by updating local copies simulating communication by using a common network mounted disk space to write each processes TTable to.

Parallelisation of PGiza++

In PGiza++ each processor processes its own sentences and calculates its own local totals and counts. At the end of the E-Step each processor dumps its TTable to disk and when all processes have finished one single process reads and ‘normalises’ (Gao’s choice of wording) the TTables. By ‘normalises’ Gao means that the counts and totals are globally reduced into one master TTable data structure (this was affirmed by email exchange with Gao directly). Then before the M-Step can proceed each process loads its own model.

Gao details an alignment task comparing the alignment speed of Giza++ on 2 processors and PGiza++ on 11 processors for the same alignment task. Giza++ took 169 hours and PGiza++ took 39 hour. This implies a 4.33x speedup. Linear speedup would be 11/2=5.5x. The parallel efficiency of this test was therefore 4.33/5.5 = 0.79. Gao observes that if we rule out the time spent in normalization, the speed up is almost linear.
Critique of PGiza++

Again, PGiza++ was a much needed step in the right direction. However, from a HPC point of view it provokes a number of criticisms:

- The use of MPI could have avoided costly writes to network storage to simulate communication.
- There is no discussion of a load balancing scheme.
- There is no suggestion that PGiza++ and MGiza++ could be merged for further speedup on hybrid architectures.
- There is no discussion of serial optimisations to Giza++. Is there even a need for parallelisation?
- There is insufficient testing of the scalability of the algorithm. It should be shown how PGiza++ performs on a number of different processors for different size data sets to the point where its limitations are clearly established.
- The observation of discounting the time spent in 'normalisation' is out of place. This is an overhead associated with the method and cannot be discounted.

In summary, PGiza++ showed some success in the speed up it provided. However, much can be done to improve on PGiza++ performance. Simulating communication by writing to common network storage is perhaps one of the slowest possible solutions to the communication needs of a parallel system. To this end Gao mentions interest in Google’s MapReduce in his concluding comments.

2.4.4 MapReduce

Introduction to MapReduce

MapReduce is not a parallelisation of EM as such. However, it is certainly worth mention. In 2004 Dean and Ghemawat presented their work on MapReduce [10] to the research community at the sixth symposium on operating system design and implementation. The relevance to EM is not immediately recognisable. However, basically speaking Dean and Ghemawat did something similar to Dempster, Laird and Rubin. They noticed that a great range of problems which were seemingly unrelated could all be handled with the same basic algorithm.

MapReduce System

Dean and Ghemawat noticed that many problem spaces could be solved by performing a mapping operation doing some work then performing a reduction operation. They further observed that such problems could be done in parallel and scale well to massively
distributed systems. The canonical example that illustrates the algorithm is counting the occurrences of words in a text file. If we split a file up into sections and then count the frequencies of words in each text file and then sum the results for each word we get exactly the same result as doing the entire operation in serial on the entire file with no chunking.

In this example the Map part of the algorithm is to Map each word to a value in a key-value data type i.e. the word is the key; its frequency is the value. We then increment the frequency by one each time a word is encountered. The reduction operation involves the totalling of all the values in local key-value pairs to corresponding values in global key-value pairs.

This class of algorithms is ideal for the parallelisation of the E-Step of EM IBM Model 1. This is because essentially the E-Step operates on key-value pairs. These are the foreign word/total key-value pair and the source word/foreign word count key-value pair. The values in the key value pairs are consistently incremented in the E-Step of IBM Model 1 and so the E-Step can be parallelised with a global reduction after working on local key-value pairs.

**Features of the MapReduce System**

There are many attractive features of the MapReduce System for developers wishing to parallelise their systems:

- The API is simple to use and the developer does not need to worry about the implementation details of the Map step or the Reduce step.

- When an application is parallelised with MapReduce the input files are automatically chunked and load balanced among processors.

- The architecture has been proven to scale well on Google’s own massively distributed system.

- The architecture is fault tolerant. No need for a developer to code fault toleration logic.

**Critique of MapReduce**

One developer’s feature can be another developer’s bug:

- The fact that the developer is alienated from the load balancing process prevents the developer from making optimisations to the load balancer. This can often be critical to high performance parallelisation.

- The fact that the developer is alienated from the implementation details of the Map step prevents the developer from making custom optimisations based on the data structures used.
• The fact that the developer is alienated from the reduction process prevents the developer from experimenting with optimisations based on reduction strategies.

• The MapReduce approach is not useful for the parallelisation of the M-Step of the EM algorithm

In summary, the MapReduce architecture is an inviting avenue for parallelising the EM algorithm for developers who are happy to get less than optimal speed up with minimal investment in development time.

2.4.5 Communication Patterns

Based on a critique of the prospect of using MapReduce to parallelise IBM Model 1 Jason Wolfe [3] considered optimising parallelisations of IBM Model 1 by using a variety of communication strategies. He considered, in addition to a MapReduce strategy, AllPairs and JunctionTree communication. He demonstrated that both communication strategies reap reductions in the communications overhead for EM IBM Model 1 especially when using larger numbers of processors.

2.5 Background Theory Summary

In summary, EM is a broadly applicable machine learning algorithm which in recent times has been extensively used for the unsupervised learning of bilingual translation dictionaries for statistical machine translation. Due to the long periods of time that researchers have to typically wait to obtain results the parallelisation of EM in various forms has been considered independently and with varying degrees of success by a number of researchers. However, no serious consideration has been given to the serial optimisation of implementations of EM such as IBM Model 1.
Chapter 3

Optimising EM - IBM Model 1

3.1 Introduction

In this chapter the general issues surrounding the design of a high performance implementation of EM for word alignment using IBM Model 1 are considered. In order to identify the parts of EM IBM Model 1 that would most benefit from optimisation and/or parallelisation the computational cost of executing the algorithm is first considered. The various sub algorithms of the algorithm are identified and which sub algorithms are the most costly is evaluated. Consideration is given to the data that must be updated in these high cost areas and experiments with different data structures are detailed.

3.2 Computational Cost of EM - IBM Model 1

3.2.1 Introduction

As explained in section 2.2.3 the EM algorithm can be regarded as an iterative two step process:

1. E-Step: Estimate complete data sufficient statistics with reference to the parameters.
2. M-Step: Select the parameters which maximise the complete data sufficient statistics

In the case of IBM Model 1 it is trivial to generate an expression of the cost of these two related processes.
3.2.2 Cost of E-Step

In calculating the cost of the E-Step of IBM Model 1 it is useful to make reference to the pseudocode:

```plaintext
set count(e|f) to 0 for all e,f
set total(f) to 0 for all f
for all sentence pairs (e_s,f_s)
   for all words e in e_s
      total_s(e) = 0
      for all words f in f_s
         total_s(e) += t(e|f)
      for all words e in e_s
         for all words f in f_s
            count(e|f) += t(e|f) / total_s(e)
            total(f) += t(e|f) / total_s(e)
```

As can be seen from casual inspection of the pseudocode the general purpose of the E-Step in IBM model 1 is to accumulate the counts and totals - the complete data sufficient statistics. This is achieved through an iterative two step process:

1. Increment the sentence local total probabilities for target words.
   - $$\text{total}_s(e) += t(e|f)$$

2. Increment the corpus wide expected counts for alignments and expected totals for source words as a function of the alignments parameters and the associated sentence local probability of the relevant target word.
   - $$\text{count}(e|f) += t(e|f) / \text{total}_s(e)$$
   - $$\text{total}(f) += t(e|f) / \text{total}_s(e)$$

Let us call the cost of one incrementation a sentence local total $$a$$, the cost of one incrementation of a corpus wide count $$b$$ and the cost of one increment of a corpus wide total $$c$$. Calculations $$a, b$$ and $$c$$ are performed for every sentence, for every target word in that sentence pair, for every source word in that target pair. In short, they are performed for every attested co-occurrence of source and target word. The cost varies from sentence to sentence as sentences vary in length. So, let $$I$$ be the variable number of source words in a sentence, let $$J$$ be the variable number of target words in the aligned sentences and $$k$$ be the number of aligned sentence pairs in the corpus. The cost can therefore be approximated as:

$$kIJ(a) + kIJ(b + c) \quad (3.1)$$
3.2.3 Cost of M-Step

In approximating the cost of the M-Step of IBM Model 1 it is, again, useful to make reference to the pseudocode:

\[
\text{for all } f \\
\quad \text{for all } e \\
\quad \quad t(e|f) = \frac{\text{count}(e|f)}{\text{total}(f)}
\]

The M-Step is very simple. The parameters, the probability of a source word given a target word, are calculated directly in a doubly nested loop looping on source words and target words. Let \( i \) equal the number of source words, \( j \) equal the number of target words and \( d \) be the cost of an update to a parameter. We can thus approximate the cost of the M-Step as:

\[ i \times j \times d \quad (3.2) \]

3.2.4 Total Cost of EM IBM Model 1

Combining equations 3.1 and 3.2 we can generate an approximate cost of one iteration of EM:

\[ kIJ(a) + kIJ(b + c) + ijd \quad (3.3) \]

Of course, this is just for one iteration. The EM algorithm is repeated until convergence.

\[ n(kIJ(a) + kIJ(b + c) + ijd) \quad (3.4) \]

And has there is an initialisation cost for initialising the parameters and sufficient statistics:

\[ n(kIJ(a) + kIJ(b + c) + ijd) + init \quad (3.5) \]

In nature, the operations \( a, b, c \) and \( d \) (the updating of the counts, totals and parameters) are similar in terms of mathematical complexity. However, \( a, b \) and \( c \) are nested one layer deeper that \( d \) and, as such, are likely to be performed more times. In order to get an idea of the relative amounts of processing time of the various parts of this formula counts were performed on French-English data from the Europarl corpus [2];

Table 3.1 is useful for an approximation of the relative execution times of the E-Step and the M-Step of a typical bilingual corpus - the French-English Europarl corpus.
The unique co-occurrences shows how many parameters need to be updated for each iteration of the M-Step. The total co-occurrences column shows how many times the complete data sufficient statistics need to be updated for each iteration of EM. It can be seen that the E-Step is far more computationally expensive than the M-Step. The speed of executing successive updates to the sufficient statistics is therefore of paramount importance to a consideration of designing a high performance implementation of this algorithm.

### 3.2.5 Summary

The E-Step and the M-step have an implied cost. Approximations of computational cost show that the E-Step is far more expensive with deeply nested reads and writes to and from data structures. Choosing data structures that permit high speed reads and writes is of paramount importance to the design a high performance implementation of the EM algorithm IBM Model 1.

### 3.3 Data Preparation and Data Structures

#### 3.3.1 Introduction

Directly related to the performance of the algorithm is the selection of data structures used to store the sufficient statistics and parameters (the counts, totals and probabilities). The speed of the implementation will largely depend on the speed of reads and writes to these data structures. The design should seek a balance between several desirable qualities of the data structures:

- Memory efficiency. Does the data structure waste memory?
• Lookup time. Does the data structure imply lookup overhead?

• Update time. Does the data structure imply update overhead?

• Spatial affinity. Are values that will be used in succession stored as neighbours in memory?

• Temporal affinity. Are values that will be reused often reused in short time intervals?

• Given the above two factors of spatial affinity and temporal affinity are the data structures designed in a way that helps the compiler perform optimisations?

• Cheap to synchronise. Does the choice of data structure imply heavy synchronisation costs for shared memory versions?

• Cheap to communicate. Does the choice of data structure imply a heavy communication cost for distributed versions?

### 3.3.2 The Dictionaries

The dictionary serves two purposes:

1. To enable the creation of digital sentences by translating from words to numeric representations of words.

2. To enable the creation of a human readable translation model by translating from numeric representation of words to the words themselves.

It is, therefore, essential that the data structure allows for the fast lookup of words given their numeric representation and the numeric representation given the word. For such kinds of lookups hash tables are ideal data structures. The alternative to a hash table could be a sorted array. With a sorted array the lookup could be performed with a binary search. However, a number of hashing algorithms (e.g. Bernstein or Fowler-Noll-Vo) are guaranteed to perform faster than a binary search with constant time.

Hash tables are made up of key-value pairs where the key is used to look up the associated value. In the case of translating words into numeric representations a hash table with words as keys and numeric identifiers as values is the ideal data structure. In the case of translating in the opposite direction, from numeric representation to word, a hash table with numeric identifiers as keys and words as values is the ideal data structure. ANSI C does not have an internal hash table implementation as other higher level programming languages such as Perl do. Uthash, an open source ANSI C compliant hash table header file available online at [19], supports a number of hashing algorithms and the ability to create hash tables with different keys using the same structures.

A key element in the design of a dictionary using the uthash header is the design of the structures that will be used as entries to the dictionary. The most fundamental components of a dictionary entry are:
1. The word itself.

2. The word’s numerical identifier.

Dictionaries are built by reading in arbitrary unseen text files which contain word strings of arbitrary length. The design of the dictionary entry structure must, therefore, be flexible enough to allow for words of any length while at the same time not wasting memory by allocating too much space for shorter words. The ideal data type for such flexible requirements is, therefore, a pointer to an array of characters of variable length.

Arbitrary text files contain an arbitrary number of different words. The design of the dictionary entry must allow enough flexibility for a large range of numeric identifiers while not wasting memory unnecessarily by using a data type which is unreasonably long. It is possible to use a selection of data types with varying ranges. While the implementation details may vary from platform to platform it is normally possible to define numeric types of length 1, 2, 4 or 8 bytes. As numeric identifiers should always be positive it makes sense to use unsigned interpretations of numeric data types in order to get the fullest possible range of values per byte. The possible unsigned ranges of usable identifiers are summarised in table 3.2 below.

<table>
<thead>
<tr>
<th>Bytes</th>
<th>From 0 to</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0 to 255</td>
</tr>
<tr>
<td>2</td>
<td>0 to 65,535</td>
</tr>
<tr>
<td>4</td>
<td>0 to 4,294,967,295</td>
</tr>
<tr>
<td>8</td>
<td>0 to 18,446,744,073,709,551,615</td>
</tr>
</tbody>
</table>

Table 3.2: This shows the possible ranges of unsigned values when using datatypes of different length to store them.

With a top end of 255 the 1 byte data type is obviously too small to be able to represent a large enough number of different words from an arbitrary file of arbitrary length. The two byte data type, with a top end of 65,535 different values, is borderline usable but could be a source of buggy behaviour when files with more than 65,535 different words are supplied as input. The four byte data type, with a top end of 4,294,967,295 different values is usable as it is unlikely that files would ever contain more different words than the range permits. The disadvantage of using this type is the need to use 4 bytes of memory for each and every data entry. The eight byte data type would be an unnecessary waste of memory. On HECToR the four byte numerical data type is used when compiling variables declared as data type int.

Taking into consideration the storage requirements of arbitrary length words, a flexible range of identifiers and the need to create a hash table in both directions a skeleton dictionary entry data structure might look like:

```c
struct dictionary_entry
{
    char *word;
};
```
unsigned int id;
    UT_hash_handle hh_word, hh_id;
};

Note the use of the UT_hash_handle data type. This is used by the uthash header for the creation and management of hash tables. As the dictionary must be addressable in both direction (from word to ID and from ID to word) two hash handles are declared in the skeleton structure. However, while data type int is 4 bytes in length on HECToR there are no guarantees that it will be so on any other platform. For portability reasons it is necessary to adapt the design of the dictionary entry structure slightly to allow for 2 byte integers on other platforms. In such cases it is necessary to use a long to achieve the same range of possible values. The adapted skeleton structure might look a little like this:

```c
struct dictionary_entry
{
    char *word;
    #if INT_IS_WORD == 1
        unsigned int id;
    #elif INT_IS_HALFWORD == 1
        unsigned long id;
    #endif
    UT_hash_handle hh_word, hh_id;
};
```

Note the use of conditional compilation instructions. It will be necessary to compile such a structure with the correct compilation flags to get a data type of the desired length. On platforms where an int is 4 bytes long it will be necessary to supply the compiler with the flag -DINT_IS_WORD=1. On platforms where a long is 4 bytes long it will be necessary to supply the compiler with the flag -DINT_IS_HALFWORD=1.

On HECToR such a structure occupies 128 bytes of memory (including packing and uthash hash table handles). This amount may vary on different platforms as different platforms may use less bytes for pointers than HECToR which uses 8 bytes (64 bit) for wider ranging addressing capabilities. Also the pointer to an array of characters word will point to a string of arbitrary length and so the reader should understand that it is not possible to predict with any degree of certainty how much memory n dictionary entries will consume for any given text.

As shall be demonstrated it is sometimes useful to store frequency information about the occurrence of words in a text. With this design feature in mind our final dictionary entry structure looks like:

```c
struct dictionary_entry
{
    char *word;
    #if INT_IS_WORD == 1
```
unsigned int id;
#elif INT_IS_HALFWORD == 1
  unsigned long id;
#endif
#if SORT == 1
#if INT_IS_WORD == 1
  unsigned int frequency;
#else INT_IS_HALFWORD == 1
  unsigned long frequency;
#endif
#endif
#if FREQ == 1
#if INT_IS_WORD == 1
  unsigned int cumulative_frequency;
#else INT_IS_HALFWORD == 1
  unsigned long cumulative_frequency;
#endif
#endif
UT_hash_handle hh_word, hh_id;
};

Not the use of the fields frequency and cumulative_frequency. Both of these fields may be included or excluded depending on how the structure is compiled (with which flags). At its maximum length on HECToR this dictionary entry data structure consumes 136 bytes of memory per entry (plus whatever string is being pointed to, of course). As shall be seen, creating, storing and using the dictionary in physical memory is fundamental to the high performance execution of the data preparation stage.

Make Dictionary Function
To build a dictionary it is necessary to execute the following steps:

1. Read in a file line by line.
2. For each line separate the sentence into individual words.
3. Check to see if the word is already in the dictionary.
4. If the item is already in the dictionary and counting is activated increment the word count for this word.
5. If the word is not in the dictionary add the new word.

In a nutshell, that is the exact sequence of logic that the function make_dictionary follows. Of course, there is memory management included as well which only allows a new dictionary item to be added if sufficient memory remains. The first step of dictionary building (reading a file in line by line) is done thanks to the operation of the fgets function:
while ( fgets ( buffer, MAX_STRING_LENGTH, input_file) 
        != NULL 
        && MAX_NO_SENTENCES > number_of_sentences ) 
{
    ... /* build dictionary logic */
}

Note the use of the constants MAX_STRING_LENGTH and MAX_NO_SENTENCES. The first is hard coded into the source and must be changed before compilation if insufficiently large. This constant is necessary as text files can have arbitrarily long sentences and fgets cannot be relied upon to read in an entire sentence. It only does so if it reaches a new-line character before the specified maximum string length is read in from the file. If this is ever exceeded, i.e. if fgets reads in an unusually long sentence and does not reach the new-line character, the program exits and lets the user know how to fix the problem. The problem can be fixed by either filtering out the long sentence or by increasing the size of the string buffer. Every effort has been made to define a buffer length which is not excessively long and yet long enough to cover even unusually long sentences. It is currently set to 1MB and this was enough to process the largest and dirtiest data set used for testing.

The second, MAX_NO_SENTENCES, is set at run time as a user specified maximum number of sentences to read in. The function is exited if this maximum is reached. Sentences are tokenised into words using the ANSI C strtok function and a whitespace separator.

cchar separators[] = " ";
token = strtok ( sentence, separators );

while ( token != NULL )
{
    /* start of new token logic */

        ... /* build dictionary logic */
        token = strtok ( NULL, separators );
}

For each token produced during sentence tokenisation the function checks if the word is already in the dictionary and if not the new word is added.

HASH_FIND ( hh_word, local_language->word2id_dictionary, 
            token, strlen ( token ), read_entry );
if ( read_entry )
{
    /* start of token already in dictionary logic */
#if SORT == 1
    read_entry->frequency += 1;
#endif
#endif
} /* end of token already in dictionary logic */
else if ( ! read_entry )
 { /* start of need to make a new entry logic */
   new_entry = calloc ( 1,
                sizeof ( struct dictionary_entry ) );
   char *this_word;
   this_word = calloc ( 1, ( strlen ( token ) + 1 ) );
   strcpy ( this_word, token );
   new_entry->id = word_id;
   new_entry->word = this_word;
   #if SORT == 1
   new_entry->frequency = 1;
   #if FREQ == 1
   new_entry->cumulative_frequency = 0;
   #endif
   #endif
   HASH_ADD_KEYPTR ( hh_word,
                   local_language->word2id_dictionary,
                   new_entry->word,
                   strlen ( new_entry->word ),
                   new_entry );
   HASH_ADD ( hh_id,
                local_language->id2word_dictionary,
                id, sizeof ( uint ), new_entry );
 }

Note the above source code snippets are a simplification of the real function which is interlaced with memory management logic. These lines of code were chosen to illustrate the basic sequence of logic of dictionary building. Note the use of pre-processor instructions to allow the dictionary entry data structure to be added with and without fields for word frequency information. Also, note the use of the HASH_FIND, HASH_ADD and HASH_ADD_KEYPTR uthash macros. These macros are used, respectively, to check if a word is already in the hash table, to add the word to the hash table with the word as the key and to add the word to the hash table with the word’s numeric identifier as the key.

Dictionary Building Experiments

To test the functionality of the make_dictionary function experiments were conducted on ten language pairs from the Europarl corpus [2]. The sentence aligned corpora were processed in their raw format and, in addition to forming two monolingual dictionaries for each language of each language pair, a running tally was made of the number of sentences, the number of words and the number of different words in each corpus.

In order to verify the correctness of the dictionary building function the word and sentence counting UNIX utility wc was used to count words and sentences in each of the
Table 3.3: Results of word and sentence counting experiments with Europarl data.

<table>
<thead>
<tr>
<th>Corpus</th>
<th>Sentences</th>
<th>Words</th>
<th>Different Words</th>
</tr>
</thead>
<tbody>
<tr>
<td>fr-en.fr</td>
<td>1,906,609</td>
<td>36,251,806</td>
<td>357,517</td>
</tr>
<tr>
<td>fr-en.en</td>
<td>1,906,609</td>
<td>34,938,170</td>
<td>264,342</td>
</tr>
<tr>
<td>it-en.it</td>
<td>1,810,308</td>
<td>32,814,373</td>
<td>377,068</td>
</tr>
<tr>
<td>it-en.en</td>
<td>1,810,308</td>
<td>34,085,591</td>
<td>260,132</td>
</tr>
<tr>
<td>es-en.es</td>
<td>1,865,373</td>
<td>35,360,832</td>
<td>359,902</td>
</tr>
<tr>
<td>es-en.en</td>
<td>1,865,373</td>
<td>34,017,495</td>
<td>261,883</td>
</tr>
<tr>
<td>pt-en.pt</td>
<td>1,847,654</td>
<td>34,285,835</td>
<td>377,019</td>
</tr>
<tr>
<td>pt-en.en</td>
<td>1,847,654</td>
<td>33,947,027</td>
<td>261,668</td>
</tr>
<tr>
<td>fi-en.fi</td>
<td>1,790,013</td>
<td>22,386,431</td>
<td>936,535</td>
</tr>
<tr>
<td>fi-en.en</td>
<td>1,790,013</td>
<td>32,472,567</td>
<td>254,948</td>
</tr>
<tr>
<td>el-en.el</td>
<td>989,046</td>
<td>17,685,421</td>
<td>347,858</td>
</tr>
<tr>
<td>el-en.en</td>
<td>989,046</td>
<td>17,816,790</td>
<td>185,898</td>
</tr>
<tr>
<td>da-en.da</td>
<td>1,862,613</td>
<td>31,130,549</td>
<td>524,184</td>
</tr>
<tr>
<td>da-en.en</td>
<td>1,862,613</td>
<td>33,828,478</td>
<td>260,687</td>
</tr>
<tr>
<td>de-en.de</td>
<td>1,877,467</td>
<td>31,785,682</td>
<td>527,346</td>
</tr>
<tr>
<td>de-en.en</td>
<td>1,877,467</td>
<td>34,224,404</td>
<td>262,609</td>
</tr>
<tr>
<td>nl-en.nl</td>
<td>1,888,571</td>
<td>34,838,385</td>
<td>411,864</td>
</tr>
<tr>
<td>nl-en.en</td>
<td>1,888,571</td>
<td>34,253,770</td>
<td>261,462</td>
</tr>
<tr>
<td>sv-en.sv</td>
<td>1,670,213</td>
<td>27,352,859</td>
<td>496,374</td>
</tr>
<tr>
<td>sv-en.en</td>
<td>1,670,213</td>
<td>29,961,346</td>
<td>246,783</td>
</tr>
</tbody>
</table>

corpora for comparison purposes. In each and every case there was agreement between `wc` and the `make_dictionary` function for word counts and sentence counts. However, this test only verifies that the two utilities processed the same number of words and sentences. It does nothing to verify the correctness of the dictionaries themselves.

In order to verify that no duplicate entries were entered in the dictionaries another function `write_dictionary` was implemented to dump the dictionaries to disk for visual inspection. For testing purposes the function was temporarily altered to only dump the words from the dictionary, one from each line. The dictionary dumps were sentence counted with `wc` then sorted with `sort` and finally any duplicate strings were removed with `uniq` and finally sentence counted a final time. In each and every case sentence counting before and after sorting and removal of duplicates verified that there were no duplicate entries in any of the dictionaries built.

In order to evaluate the computational and memory consumption requirements of the `make_dictionary` function a number of timing experiments were performed on dictionary building runs on the same ten language pairs of the Europarl corpus. All timings were taken on unoptimised versions of the function compiled with the debug flag `-g` and on optimised versions compiled with the `-fastsse` flag.
Table 3.4: Memory consumption and time to build of dictionaries for Europarl corpus data.

### 3.3.3 Digital Sentences

Fundamental to the fast processing of EM IBM Model 1 is to keep key data structures in memory in forms which are simple and fast to access. To this end the in memory representation of sentences is perhaps the most significant of data structures and its design can have a huge impact on the performance of IBM Model 1. The function `make_digital_sentences` performs the following functions:

1. It reads in a text file line by line in similar fashion to the `make_dictionary` function.
2. It tokenises sentences on the white-space separator to process individual words.
3. For each word it retrieves the word’s numerical ID from the dictionary hash table.
4. It produces an in-memory representation of each word of each sentence by using its numerical ID.

There are several ways that sentences could be represented in memory. The most obvious way is to simply load their ASCII character representations into memory and operate on those. However, as words are variable length this has complexity implications for the processing of in-memory sentences while performing EM. On each iteration of EM the sentence would have to be tokenised again as the words are of variable length. Also, the use of ASCII representations implies being limited to data structures which are addressable by the ASCII strings themselves for the translation table. This would exclude, or make extremely complicated, optimal solutions like a simple 2D array for the translation table.

GIZA++ takes as input and processes numeric sentences. There are clear advantages to such an approach. Once the decision has been made as to which numeric data type to use each token in a sentence will be of fixed length and there will be no need for any kind of tokenisation. Also, numeric representations of words can be much smaller.
than ASCII equivalents. For example, when using an `int` data type on HECToR the implications are that the numeric representation is shorter than every string of four or more characters (accounting for the NULL terminator of each string).

Having opted for a numeric representation of sentences and made the design choice that they should be kept in-memory for performance purposes there remain a couple of obvious solutions for the exact data structure of the in-memory digital sentences. As sentences can be of variable length one possible solution could be to store a 1D array of pointers to integer arrays - one pointer for each sentence. The implications are that sentences may not necessarily be contiguous in memory as when they are allocated they will be assigned space on the heap. Such a strategy also implies the need to store the length of each digital sentence or mark the end of each sentence in some way such that they can be processed without running the risk of causing bugs by running off the end of a digital sentence and processing random locations in memory.

The selected solution was to store digital sentences as one contiguous 1 dimensional array of unsigned integers with sentences separated by a 0 valued integer. This strategy was also backed up with a separate one dimensional array of unsigned integers with one element for each sentence. Each element of this array stores the numerical offset for the beginning of the corresponding digital sentence in the digital sentence array. The data structure for storing digital sentences along with their index array (and other information) looks like this:

```c
struct language
{
    unsigned int number_of_words;
    unsigned int number_of_different_words;
    unsigned int number_of_sentences;
    unsigned int *sentences;
    unsigned int *sentence_index;
    struct dictionary_entry *word2id_dictionary;
    struct dictionary_entry *id2word_dictionary;
};
```

Note that the data structure, as well as including pointers to arrays of unsigned integers for the digital sentences and their index, also has pointers for the dictionary hash tables and for the sake of convenience variables to store the number of sentences, the number of words and the number of different words found in a corpus. Keeping this information together in the same structure for any particular language simplifies the readability of the code.

The main logic of the `make_digital_sentences` function can be seen in the following code snippet:

```c
copy ( sentence, buffer );

token = strtok ( sentence, separators );
```
The flow of logic is very similar to that of the make_dictionary function. The ASCII text is read in one line at a time from the hard disk. The sentences are tokenised into individual words. And then the ID of each word is retrieved so that it can be placed in the next available space in the sentences array. At the end of each sentence a 0 value is stored to mark the division between the sentences. The use of a 0 is convenient as IBM Model 1 can make use of a 0 value for the NULL alignment (when words have no translation in the aligned sentence).

Here are some English sentences from the Europarl corpus and their digital equivalents when using an unsorted dictionary:

<chapter id=1>
resumption of the session
<speaker id=1 name="president">
i declare resumed the session of the european parliament adjourned on friday 17 december 1999 , and i would like once again to wish you a happy new year in the hope that you enjoyed a pleasant festive period .
<p>
you have requested a debate on this subject in the course
of the next few days, during this part-session. Please rise, then, for this minute’s silence.

3.4 EM and The Translation Table (Counts, Totals and Probabilities)

All counts totals and probabilities require floating point accuracy. On HECToR there are two choices of floating point number float (4 bytes) and double (8 bytes). There are advantages and disadvantages to the use of either. The use of a float data type implies lighter memory usage and faster processing at the expense of accuracy. The double data type improves on the float data type in terms of accuracy at the expensive of doubly heavier memory usage and more intensive processing requirements.

In the E-Step counts and totals are updated arbitrarily. These arbitrary updates to counts and totals constitute the lion’s share of the computational cost of EM with IBM Model 1. It is, therefore, of paramount importance that the selected data structure allows for such arbitrary updates and is optimised repeated updates in close temporal proximity.

3.4.1 2D Array - Matrix

For the totals a 1 dimensional array of floating point numbers will do as totals are stored in monolingual fashion. The most obvious solution for the storage of the counts and probabilities is a 2 dimensional array with the bounds of the first dimension corresponding to source word ID’s and the bounds of the second corresponding to target word ID’s. In this way it will be possible to arbitrarily address a count with the source word ID and the target word ID.

However, while in theory a 2 dimensional array could be ideal for high performance updates of counts in practise this approach has its limitations. For example, a test corpus of 100,402 source words and 80,127 target words would require a 2 dimensional array
of 100,402 * 80,127 elements. That is 8,044,911,054 entries. If storing one float for each count and one float for each probability that would be 8,044,911,054 * 8 bytes. That is 64,359,288,432 bytes or about 60 GB. If using doubles about 120GB of physical memory would be needed. Each node on HECToR is limited to only 8GB of physical memory and so storing the entire table in memory on a single node is impractical.

However, interesting results can be obtained by processing a selected subset of this table. A function, make_alignment_table, which builds the largest 2 dimensional array of structures memory limitations will allow. The structures used to store entries in the alignment table were defined as follows:

```c
struct matrix_translation
{
    #if PRECISION == 1
        float count;
        float probability;
    #endif
    #if PRECISION == 2
        double count;
        double probability;
    #endif
};
```

Notice the use of pre-processor instructions to conditionally compile the entries in support of both float and double data types. The make_alignment_table function makes a binary search to discover how much memory is available. For HECToR, which is configured with no swap space, relying on malloc or calloc to determine if physical memory is available can work fine. However, for machines configured with swap space these functions give no indication of how much actual physical memory is available as they allocate virtual memory and not actual physical memory. For this reason, in addition to using calloc for memory allocation the make_alignment_table function checks that memory usage will not exceed user defined bounds. Relying on the user to let the function know how much physical memory is available can also be hazardous. However, a function that checks if real physical memory is available would require root privileges and non-trivial system level programming.

The function first recursively divides the number of source words and target words in the corpus by two until it finds an array which would fit in the amount of memory the user said it was safe to use:

```c
while ( source_language->size > 1
    && target_language->size > 1 )
{
    mem_required = source_language->size
        * sizeof ( struct matrix_translation * );
    mem_required += source_language->size
        * target_language->size
```
* sizeof ( struct matrix_translation );
if ( mem_required > mem_left )
{
    source_language->size /= 2;
    target_language->size /= 2;
}
else
{
    break;
}
}

Then it attempts to allocate memory for those discovered dimensions. If memory allo-
cation fails another binary search is performed to find dimensions the system will allow
the function to allocate.

while ( source_language->cache_size > 1
    && target_language->cache_size > 1 )
{
    tmp_translation_table_d1 = calloc ( source_language->size,
                              sizeof ( struct matrix_translation * ) );
    tmp_translation_table_d2 = calloc ( source_language->size
                              * target_language->size,
                              sizeof ( struct matrix_translation ) );
    if ( tmp_translation_table_d2 == NULL
         || tmp_translation_table_d1 == NULL )
    {
        source_language->size /= 2;
        target_language->size /= 2;
    } /* End of memory allocation failed section */
else
{
    break;
}
}

free ( tmp_translation_table_d1 );
free ( tmp_translation_table_d2 );

Once a table has been found that both the user and the system agree can fit in memory
the function recursively grows that table with another binary search. This is done by
incrementing the table’s dimensions by 50% of the last attempt to grow the table. The
first iteration attempts to grow the table by 50% of its current dimensions:

source_inc = source_language->size / 2;
target_inc = target_language->size / 2;
while ( source_inc > 1 && target_inc > 1 )
{
    ... check that new dimensions fit in user specified
    ... memory and that memory allocation goes fine
    {
        source_language->size += source_inc;
        target_language->size += target_inc;
        free ( tmp_translation_table_d2 );
        free ( tmp_translation_table_d1 );
    }
}

When using reduced dimensions it is necessary to filter the digital sentences such that only sentences containing words in the bounds of the table are processed. As EM is typically used in multiple iterations it makes good sense to filter the sentences before passing them to EM such that the overhead of filtering be incurred only once. The function filter_sentences was made for this very purpose.

Using a French-English corpus of 1,555,073 aligned sentences with 100,402 different French words and 80,127 different English words and a memory limit of 7GB it was possible to construct a naïve table of dimensions 33,754 * 26,939. Filtering the sentences on these dimensions rendered 72% of the sentences (an unexpectedly large proportion) and 51% of the actual attested alignments. The dump of the first 3000*3000 elements of this naïve table shows why.

The implied matrix is an extremely sparse matrix as many source language words, especially rarely occurring words, do not co-occur in sentences with all the target language words from the corpus. The dump in figure 3.1 shows black for where source language words and target language words co-occur and white where they do not. As can be seen in figure 3.1 there is quite a dense area in the upper left hand corner of the matrix. This is because words which happen frequently are found quite early on in a corpus, even with such a naïve approach.

A serial version of EM IBM Model 1 was tested on HECToR on these filtered sentences for five iterations with the following results:

<table>
<thead>
<tr>
<th>Timed section</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Time</td>
<td>121.32</td>
</tr>
<tr>
<td>Data preparation</td>
<td>20.91</td>
</tr>
<tr>
<td>EM</td>
<td>100.42</td>
</tr>
<tr>
<td>E-Step</td>
<td>10.10</td>
</tr>
<tr>
<td>M-Step</td>
<td>4.37</td>
</tr>
</tbody>
</table>

Table 3.5: Naive matrix performance.

The main limitation of the proportion of sentences that can be processed using such
Figure 3.1: Naive distribution of co-occurrences. y axis = source words, y axis = target words
a naïve approach is the amount of sparsity in the reduced table used. By making the reduced table more dense it is possible process more sentences while using the same size of reduced table. A simple, and computationally inexpensive, mechanism for making the reduced matrix more dense is to sort the dictionaries by frequency before creating the digital sentences. By assigning the most frequent words the lowest values the laws of statistics dictate that the most frequent co-occurrences will implicitly make their way into the upper left hand corner of the alignment table. Figure 3.2 confirms this. It shows the distribution of co-occurring word pairs in a reduced table built with the same limitations as that built in naïve fashion. The only difference is that the in-memory dictionaries were sorted by order of descending frequency and numerical identifiers were reassigned before the creation of the digital sentences. As can be seen there is a very dense area in the top left corner of the table (black indicates points in the matrix which correspond to real co-occurrences).

<table>
<thead>
<tr>
<th>Timed section</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Time</td>
<td>139.18</td>
</tr>
<tr>
<td>Data preparation</td>
<td>21.33</td>
</tr>
<tr>
<td>EM</td>
<td>117.85</td>
</tr>
<tr>
<td>E-Step</td>
<td>14.28</td>
</tr>
<tr>
<td>M-Step</td>
<td>4.59</td>
</tr>
</tbody>
</table>

Table 3.6: Dense matrix performance.

As can be seen in table 3.6 there was a very small increase in the data preparation overhead (0.42 seconds). However, the benefits of this increased overhead were that it was possible to process 90% of the sentences in the corpus rather than 72% and 75% of the alignments rather than 51%. A small overhead with justifiable benefits. For a 6.9% increase in overhead it was possible to almost 1.5x as much data. As the E-Step was processing more alignments in the denser matrix it comes of no surprise that the E-Step took longer (14.28 seconds instead of 10.90). There was little difference in the time taken to execute the M-Step.

It should be kept in mind that these times come at the cost of working on a reduced alignment table which does not cover all of the attested alignments in the corpus. However, there may be good justification for this as Brown et al showed that low frequency words noisy data and be damaging to the results of EM. This is, in fact, generally true of the EM algorithm and there may be good reason for filtering out low frequency data for other applications of EM.

In spite of all this some researchers may wish to be able to use the full data set with no filtering. In such cases the main disadvantage of using a 2 dimensional array comes to the fore. It wastes a lot of memory on non-attested alignments of words which could never be translations of each other.
Figure 3.2: Condensed distribution of co-occurrences. y axis = source words, y axis = target words
3.4.2 Hash Table

The problem of how to store all alignments in memory while not incurring excessive overhead on the EM part of the program is an interesting one. As already shown the speed of the EM IBM Model 1 implementation largely depends on the speed that counts, totals and probabilities can be updated. Storing the totals in a 1 dimensional array is not wasteful of memory and so there was no need to change this optimal data structure.

For the counts and probabilities one option could be to store them in a sorted 1 dimensional array and carry out lookups via a binary search of the array. However, as hashing algorithms exist which are generally faster than binary searches a hash table was implemented and it was possible to fit the whole alignment table in memory (along with dictionaries and digital sentences) using less than 3GB of memory. Of course, it is to be expected that EM IBM Model 1 will perform much more slowly on such a data structure as read and write to the alignment table will require waiting for the hashing algorithm to complete a lookup.

The function `make_alignment_hash_table` was implemented to create the hash table. The definition of the data structure for entries to the hash table, allowing for both `float` and `double` data types, looks like this:

```c
struct hash_translation
{
    #if INT_IS_WORD == 1
        unsigned int source_word;
        unsigned int target_word;
    #elif INT_IS_HALFWORD == 1
        unsigned long source_word;
        unsigned long target_word;
    #endif
    #if SORT == 1
        #if INT_IS_WORD == 1
            unsigned int frequency;
        #if FREQ == 1
            unsigned int cumulative_frequency;
        #endif
        #elif INT_IS_HALFWORD == 1
            unsigned long frequency;
        #if FREQ == 1
            unsigned long cumulative_frequency;
        #endif
    #endif
    #endif
    #if PRECISION == 1
        float count;
        float probability;
    #endif
    #endif
    #endif
    #endif
};
```
The function loops through the digital sentences and through each source language word in each sentence and finally through each target language word in the corresponding target language sentence. In this way the function finds all attested co-occurrences in the corpus:

```c
for ( sentence_number = 0 ;
    sentence_number < source_language->number_of_sentences;
    sentence_number++ )
{
    for ( s = source_language->sentence_index[ sentence_number ];
        s < source_language->sentence_index[ sentence_number+1 ];
        s++ )
    {
        source_token = source_language->sentences[s];

        t = target_language->sentence_index[ sentence_number ];

        for ( t = target_language->sentence_index[ sentence_number ];
            t < target_language->sentence_index[ sentence_number+1 ];
            t++ )
        {
            target_token = target_language->sentences[t];
            ... build the hash table
        }
    }
}
```

In order to preserve the uniqueness of entries to the hash table on each encounter of a co-occurrence a lookup to the hash table is performed to see if the entry already exists in the hash table. If not, a new entry is added. The following code snippet shows the basic flow of logic of the function. For brevity memory management sections have been omitted:

```c
struct hash_translation *search_result;
lookup_key.source_word = source_token;
lookup_key.target_word = target_token;
key_length = offsetof ( struct hash_translation,
                       target_word)
```
HASH_FIND ( hh, translations,
    &lookup_key.source_word,
    key_length, search_result );

if ( search_result )
{
    #if SORT == 1
        search_result->frequency += 1;
    #endif
}
else
{
    struct hash_translation *this_translation;
    this_translation = malloc ( sizeof
        ( struct hash_translation ) );
    memset ( this_translation,
        0,
        sizeof ( this_translation ) );

    if ( this_translation )
    {
        mem_left -= sizeof ( struct hash_translation );
        mem_used += sizeof ( struct hash_translation );
        memset ( this_translation, 0,
            sizeof ( this_translation ) );
        this_translation->target_word = target_token;
        this_translation->source_word = source_token;
        #if SORT == 1
            this_translation->frequency = 1;
        #endif
        key_length = offsetof ( struct hash_translation,
            target_word
                + sizeof ( unsigned int )
                - offsetof ( struct hash_translation,
                    source_word));

        HASH_ADD( hh, translations, source_word,
            key_length, this_translation );

        Operating on a hash table the implementation of EM IBM Model 1 completed with the following results.
<table>
<thead>
<tr>
<th>Timed section</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Time</td>
<td>1264.58</td>
</tr>
<tr>
<td>Data preparation</td>
<td>280.26</td>
</tr>
<tr>
<td>EM</td>
<td>984.33</td>
</tr>
<tr>
<td>E-Step</td>
<td>191.24</td>
</tr>
<tr>
<td>M-Step</td>
<td>1.38</td>
</tr>
</tbody>
</table>

Table 3.7: Hash table performance.

The cost of being able to process all of the sentences in the corpus is marked. The overhead of the data preparation stage has increased more than ten fold. The increased overhead in the data preparation stage is due to the creation of the hash table which must do a lookup for each and every occurrence of a source language word / target language word pair.

In this test 577,724,989 alignments were processed only 33,007,041 of which resulted in a new addition to the hash table. Quite plainly, the majority of lookups while building the hash table do not contribute to growing the hash table. One way of speeding up the process of building the hash table may be to create a 1 dimensional array of alignments from the corpus which is then sorted and to remove duplicates leaving a list of unique alignments which can be added to the hash table in rapid succession with no need to check if they are already in the hash table.

With respect to the matrix versions the M-Step with a hash table runs faster. With this test setup the M-Step from 4.59 seconds (dense matrix) to 1.38. This is because hash tables implemented with ut-hash are also doubly linked lists. This means they can be stepped through sequentially as the following code snippet shows:

```c
for (search_result=translations;
    search_result!=NULL;
    search_result=search_result->hh.next)
{
    if ( source_language->totals
        [search_result->source_word] > 0 )
    {
        search_result->probability =
        search_result->count
        / source_language->totals[search_result->source_word];
    }
}
```

However, the E-Step with the hash table is considerably more computationally expensive rising from 14.28 seconds (dense matrix) to 191.24 seconds. This is 13.38 times more expensive. Of course, the hash table version is processing more alignments but the main cause of this computational overhead is the need to wait for hashing functions to return the address of the alignment to be read or updated. This optimal use of the
cache and direct addressing is preferable from this perspective.

Both the dense matrix version and the hash table version have desirable features. The dense matrix version processes frequently occurring co-occurrences making good use of cache optimisation. The hash table version is capable of storing and processing more alignments using less space in-memory. A hybrid version was, therefore, implemented in an attempt to reap the benefits of both structures.

### 3.4.3 Hybrid - Hash Table with Cache Table

The `make_alignment_table` function was adapted to enable the building of a software implemented cache table of pointers to entries in the hash table. The pointers in the cache table are linked to entries in the hash table using the logic illustrated in the following code snippet:

```c
for ( search_result=translations;          
     search_result!=NULL;                        
     search_result=search_result->hh.next )
{
    s = search_result->source_word;        
    t = search_result->target_word;        
    if ( s < source_language->cache_size &&    
         t < target_language->cache_size )
    {
        cache_table[s][t] = search_result;            
        linked++;
    }
}
```

The hybrid version was tested on the same data set with the same 7GB memory limitation with the following results.

<table>
<thead>
<tr>
<th>Timed section</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Time</td>
<td>848.55</td>
</tr>
<tr>
<td>Data preparation</td>
<td>483.19</td>
</tr>
<tr>
<td>EM</td>
<td>365.35</td>
</tr>
<tr>
<td>E-Step</td>
<td>62.40</td>
</tr>
<tr>
<td>M-Step</td>
<td>1.47</td>
</tr>
</tbody>
</table>

Table 3.8: Hybrid hash table with cache table performance.

As can be seen in table 3.8 while there was a massive increase (over 200 seconds) in the time taken for data preparation the overall processing time was reduced from 1264.58 seconds to 848.55. This is largely due to the speed up of the E-Step from 191.24 seconds.
to 62.40 seconds. However, standing at 483.19 seconds (over half of the total operation time) it is clear that the overhead of the data preparation stage is a bottleneck to be taken seriously.

In an attempt to reduce this bottleneck the same experiment was repeated on the same data set but with a 4GB memory limitation to see what effect reducing the size of the cache table has on performance. The results were as follows:

<table>
<thead>
<tr>
<th>Timed section</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Time</td>
<td>586.97</td>
</tr>
<tr>
<td>Data preparation</td>
<td>287.66</td>
</tr>
<tr>
<td>EM</td>
<td>299.31</td>
</tr>
<tr>
<td>E-Step</td>
<td>56.54</td>
</tr>
<tr>
<td>M-Step</td>
<td>1.39</td>
</tr>
</tbody>
</table>

Table 3.9: Hybrid hash table with cache table performance.

As can be seen in table 3.9 the total time, the data preparation time, the EM time and the E-Step execution time all improved as a result of reducing the size of the cache table. The data preparation time was clearly reduced because of the need to link less entries in the cache table.

The EM execution time improved as a result of the improvement of the E-Step execution time. The E-Step execution time may have improved because of better cache usage. In any case, the implications are clear that there exists an optimal size of cache table which taking into account the additional time for data preparation and the improvements gained from reduced E-Step execution time will give the best overall performance. A simple heuristic for calculating the size of the optimal cache table remains elusive but may, in some way, be related to the cumulative frequency of the most frequent collocations.

### 3.4.4 Summary

In summary, it has been shown that a 2 dimensional matrix is the optimal data structure for a serial implementation in terms of performance. However, such a data structure is limited by memory constraints which can be overcome, to an extent, by creating a dense quadrant in the upper left corner of the matrix.

In terms of memory usage a hash table implementation was able to store counts and probabilities for far more co-occurrences using less memory. This comes at the price of a performance hit. It was shown that this performance hit can be eased by the use of a dense matrix as a software cache table. The size of this cache table can have effects on the overall performance. A simple heuristic for calculating the optimal size of software cache table remains elusive.

Overall results and speedups relative to the slowest version (the hash table version) are summarised in table 3.10
<table>
<thead>
<tr>
<th>Version</th>
<th>Time (seconds)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash</td>
<td>1264.58</td>
<td>1</td>
</tr>
<tr>
<td>Naïve matrix</td>
<td>121.32</td>
<td>10.42</td>
</tr>
<tr>
<td>Dense matrix</td>
<td>139.18</td>
<td>9.09</td>
</tr>
<tr>
<td>Hybrid 1</td>
<td>848.55</td>
<td>1.49</td>
</tr>
<tr>
<td>Hybrid 2</td>
<td>586.97</td>
<td>2.15</td>
</tr>
</tbody>
</table>

Table 3.10: Hybrid hash table with cache table performance.
Chapter 4

Parallel EM

4.1 Shared Memory EM IBM Model 1

A shared memory implementation was conceived by working on a shared data structure for the translation table. Counts and totals are synchronised in critical sections to ensure no corruption of end results. However, because of the high level of serial optimisation such a solution proved less than optimal. Table 4.1 shows times for a well load balanced implementation with no synchronisation (synchronisation was intentionally disabled to demonstrate the negation of cache optimisations).

<table>
<thead>
<tr>
<th>Processors</th>
<th>Time (seconds)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20.13</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>16.74</td>
<td>1.20</td>
</tr>
<tr>
<td>3</td>
<td>12.59</td>
<td>1.60</td>
</tr>
<tr>
<td>4</td>
<td>10.82</td>
<td>1.95</td>
</tr>
<tr>
<td>5</td>
<td>9.97</td>
<td>2.11</td>
</tr>
</tbody>
</table>

Table 4.1: Shared memory timings with synchronisation disabled on dense matrix version.

As can be seen in table 4.1 the speedup, even without synchronisation, is much less than linear. The speedup using 2 processors was only 1.20. On 3 processors only 1.60. The reader should bear in mind that this is without synchronisation overhead (i.e. On an incorrect implementation). The purpose of the experiment is to show that because of the good use of cache in matrix versions using a shared memory implementation negates such optimisations for reasons of problems with cache coherency. That is to say that each processor, having its own cache, is no longer able to use cached entries when another processor has updated the corresponding entry in the translation table.
4.2 Distributed EM IBM Model 1

The matrix versions were parallelised with MPI. In the MPI version each processor works on a local copy of the translation table. Each processor performs its own local E-Step and then before going on to perform the M-Step all the local translation tables are synchronised by performing a global reduction operation via the AllReduce mechanism which globally reduces to a master processor and then distributes the globally reduced counts and totals in broadcast fashion to all processors. The following code snippet illustrates:

```c
#if MPI == 1
if ( number_of_tasks > 1 )
{
    double send_buffer[target_language->size];
    double receive_buffer[target_language->size];

#if DATATYPE == 1 || DATATYPE == 2
    for ( s = 0; s < source_language->size; s++ )
    {
        /*
         * Copy a row of the table to a buffer
         */
        for ( t = 0; t < target_language->size; t++ )
        {
            send_buffer[t] = translation_table[s][t].count;
        }
        /*
         * Globally reduce the row in the buffer and receive a
         * to the receive buffer
         */
        MPI_Allreduce ( send_buffer, receive_buffer,
                        target_language->size, MPI_DOUBLE,
                        MPI_SUM, MPI_COMM_WORLD );

        /*
         * Update the translation table from the
         * receive buffer
         */
        for ( t = 0; t < target_language->size; t++ )
        {
            translation_table[s][t].count = receive_buffer[t];
        }
    }
#endif
#endif
```
As the results in table 4.2 show any speedup of the E-Step is negated by the communication costs of the AllReduce operation when testing using the English-French Europarl corpus.

<table>
<thead>
<tr>
<th>Processors</th>
<th>E-Step(seconds)</th>
<th>C-Step</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5.92</td>
<td>0.00</td>
</tr>
<tr>
<td>2</td>
<td>2.89</td>
<td>3.91</td>
</tr>
<tr>
<td>3</td>
<td>1.73</td>
<td>6.55</td>
</tr>
</tbody>
</table>

Table 4.2: MPI timings of E-Step and C-Step using dense matrix version on English-French Europarl corpus.

This is in stark contrast to the speedup gained by Wolfe [3] when using a similar strategy on up to 20 different nodes. However, as Wolfe’s time per iteration on a 2 million sentence pair dataset was over 250 seconds compared with 14.28 seconds per iteration of the dense matrix version on just over 1.5 million sentence pairs this may be explained by the fact that proportionally speaking the E-Step is not running long enough for the
cost of communication to be worthwhile. Further testing with larger data sets may show that an MPI parallelisation of the dense matrix implementation is worthwhile.

Also, there exists the possibility of reducing communication costs by only sending non-zero quantities in the C-Step. This would imply extra overhead in the preparation to communicate and in the interpretation of communicated results but may be worth investigating. Unfortunately, there was not sufficient time to experiment any further.
Chapter 5

Conclusions

It has become clear that before considering a parallelisation strategy the question must be asked if speedup can be obtained from serial optimisations. In the special case of IBM Model 1 it has been possible to highly optimise a serial version by exploiting the nature of the input data and arranging a 2 dimensional matrix such that the most common words have the lowest ID’s such that a highly dense quadrant can be created in the upper left hand corner of the matrix. Using this statistical trick it was possible to fit a large number of co-occurrences in memory and be able to benefit from the direct addressing capabilities of this data structure. As the implementation of IBM Model 1 has good temporal proximity of reads and updates for counts and totals it was possible to make good use of cache memory.

It has also been shown that it is possible to speedup a hash table implementation by the use of a dense matrix software cache table. As there is extra overhead associate with the setup of this software cache table the size of the cache table can affect the amount of speedup gained. A simple heuristic for automatically generating a cache table of optimal size remains elusive but may be related to the cumulative frequency of the most frequent co-occurrences.

There is still much that can be done in optimising a serial version of IBM Model 1 and this should be explored to its very depths before considering parallelisation.

Now that a dense matrix implementation of IBM Model 1 has been realised that can complete on the English-French Europarl corpus in 139 seconds plans are in progress to use the algorithm to perform a brute force search for the global maximum by setting off the algorithm with different initialisations such that each instance may converge on a different local maximum.
Bibliography


