Democratic and Popular Republic of Algeria
Ministry of Higher Education and Scientific Research
Ahmed Draia University - Adrar
Faculty of Science and Technology
Department of Mathematics and Computer Science

A Thesis Presented to Fulfil the Master's Degree in Computer Science

Option: Intelligent Systems.

Title:

Distributed Memory Bound Word Counting For Large Corpora

Prepared by:
Bekraoui Mohamed Lamine & Sennoussi Fayssal Taqiy Eddine

Supervised by:
Mr. Mediani Mohammed

In front of

- President : CHOUGOEUR Djilali
- Examiner : OMARI Mohammed
- Examiner : BENATIALLAH Djelloul

Academic Year 2017/2018

Abstract:

Statistical Natural Language Processing (NLP) has seen tremendous success over the recent years and its applications can be met in a wide range of areas. NLP tasks make the core of very popular services such as Google translation, recommendation systems of big commercial companies such Amazon, and even in the voice recognizers of the mobile world.

Nowadays, most of the NLP applications are data-based. Language data is used to estimate statistical models, which are then used in making predictions about new data which was probably never seen. In its simplest form, computing any statistical model will rely on the fundamental task of counting the small units constituting the data.

With the expansion of the Internet and its intrusion in all aspects of human life, the textual corpora became available in very large amounts. This high availability is very advantageous performance-wise, as it enlarges the coverage and makes the model more robust both to noise and unseen examples. On the other hand, training systems on large data quantities raises a new challenge to the hardware resources, as it is very likely that the model will not fit into main memory. This is where the external memory (disk) comes in handy.

The idea of exploiting the disks to hold parts of the data while working on other parts in main memory proved very useful in many NLP tasks. In particular, the tasks which may inflate the training data with combinatorial compositions, such as machine translation or language modelling. However, to the best of our knowledge, the solutions which exist for this purpose do not take full advantage of the parallel architectures which are very common nowadays.

This work is an initial step towards combining parallelism with external memory in the n-gram counting task. We build our solution on top of the STXXL library, which gives access to many ready-to-use external memory data structures, while supporting parallelism in the IO level. We go a step further, by distributing the computation over many machines each of which runs the computation using multiple threads. The result is a highly parallel solution, which can perform counting on large datasets on limited resources hardware. Our solution can achieve up to 4 times speedup over a single unit-single disk solution.

To the best of our knowledge, software products which perform this task (e.g.KenLM) have the following limitations: assuming that the vocabulary is not larger than the ram, don’t support distributed parallelism and don’t support parallel disk model.

Our work some with the objective to overcome these limitations by indexing the text wish give us the ability to support large vocabulary , use STXXL wish give us the ability to support parallel disk model , use different component to support distributed parallelism.

This thesis describes the several performance techniques used to apply our solutions.

Key words: Memory, NLP, STXXL, Corpora, disk

Résumé:

Le statistique Traitement Automatique de la Langue Naturelle (TALN) a connu un succès considérable au cours des dernières années et ses applications peuvent être satisfaites dans un large éventail de domaines. Tâches de la (TALN) font le noyau de services très populaires tels que la traduction Google, les systèmes de recommandation des grandes entreprises commerciales telles que Amazon, et même dans les connaisseurs de la voix du monde mobile.

De nos jours, la plupart des applications (TALN) sont basées sur des données. Les données linguistiques sont utilisées pour estimer les modèles statistiques, qui sont ensuite utilisés pour faire des prédictions sur de nouvelles données qui n'ont probablement jamais été vues. Dans sa forme la plus simple, le calcul d'un modèle statistique reposera sur la tâche fondamentale de compter les petites unités constituant les données.

Avec l'expansion d'Internet et son intrusion dans tous les aspects de la vie humaine, les corpus textuels sont devenus disponibles en très grande quantité. Cette haute disponibilité est très avantageuse en termes de performances, car elle élargit la couverture et rend le modèle plus robuste à la fois pour le bruit et les exemples non vus. D'un autre côté, la formation de systèmes sur de grandes quantités de données soulève un nouveau défi pour les ressources matérielles, car il est très probable que le modèle ne rentrera pas dans la mémoire principale. C'est là que la mémoire externe (disque) est utile.

L'idée d'exploiter les disques pour conserver des parties des données tout en travaillant sur d'autres parties de la mémoire principale s'est avérée très utile dans de nombreuses tâches (TALN). En particulier, les tâches qui peuvent gonfler les données d'apprentissage avec des compositions combinatoires, telles que la traduction automatique ou la modélisation de langage. Cependant, à notre connaissance, les solutions qui existent à cet effet ne tirent pas pleinement parti des architectures parallèles qui sont très courantes de nos jours.

Ce travail est une première étape vers la combinaison du parallélisme avec la mémoire externe dans la tâche de comptage n-gramme. Nous construisons notre solution en se basant sur la bibliothèque STXXL, qui donne accès à de nombreuses structures de données de mémoire externe prêtes à l'emploi, tout en prenant en charge le parallélisme au niveau des E / S. Nous allons plus loin, en répartissant le calcul sur de nombreuses machines dont chacune exécute le calcul en utilisant plusieurs threads. Le résultat est une solution hautement parallèle, qui peut effectuer le comptage sur de grands ensembles de données sur du matériel à ressources limitées. Notre solution peut atteindre jusqu'à 4 fois l'accélération sur une seule unité de disque unique.

À notre connaissance, les produits logiciels qui exécutent cette tâche (par exemple, KenLM) ont les limitations suivantes: le vocabulaire n'est pas plus grand que la RAM, ne supportant ni le parallélisme distribué, ni le modèle de disque parallèle.

Notre travail dans l'objectif de surmonter ces limitations en indexant le texte, nous permet de prendre en charge un large vocabulaire, l'utilisation de STXXL nous permet de prendre en charge des modèles de disques parallèles, d'utiliser différents composants pour supporter le parallélisme distribué.

Cette thèse décrit les différentes techniques de performance utilisées pour appliquer nos solutions.

Clés : mémoire, TALN, STXXL, corpus, disque

الملخص:

لقد شهدت معالجة اللغات الطبيعية الإحصائية نجاحًا هائلاً خلال السنوات الأخيرة، ويمكن رؤية تطبيقاتها في مجموعة واسعة من المجالات. إن مهام المعالجة الآلية للغة هي أساس الخدمات المشهورة للغاية مثل ترجمة 'جوجل' وأنظمة التوصية الخاصة بالشركات التجارية الكبرى مثل 'أمازون'، وحتى في أجهزة التعرف على الصوت في عالم الهواتف المحمولة.

في الوقت الحاضر، معظم تطبيقات المعالجة الآلية للغة تعتمد على البيانات. يتم استخدام بيانات اللغة لتقدير النماذج الإحصائية، والتي يتم استخدامها بعد ذلك في إجراء تنبؤات حول البيانات الجديدة التي ربما لم يسبق رؤيتها. في أبسط الحالات، يعتمد حساب أي نموذج إحصائي على المهمة الأساسية المتمثلة في عدَ الوحدات المشكلة للبيانات.

مع التوسع في الإنترنت وتدخلها في جميع جوانب الحياة البشرية، أصبحت النصوص متاحة بكميات كبيرة جدا. هذا الكم الهائل هو شيء مفيد للغاية، لأنه يوسع التغطية ويجعل النموذج أكثر قوة ومقاومة للتشويش والأمثلة غير المرئية. من ناحية أخرى، تواجه هذه الأنظمة التي تحتوي على كميات كبيرة من البيانات تحديًا جديدًا في الجهاز الحاسب، حيث من المحتمل جدًا ألا يتناسب حجم النموذج مع الذاكرة الرئيسية. وهاهنا يُفعل دور الذاكرة الخارجية (القرص).

أثبتت فكرة استغلال الأقراص لحفظ بعض الأجزاء من البيانات أثناء العمل على أجزاء أخرى في الذاكرة الرئيسية أنها مفيدة للغاية في العديد من تطبيقات معالجة اللغة. على وجه الخصوص، المهام التي قد تضخم بيانات التدريب لكثرة التركيبات، مثل الترجمة الآلية أو نمذجة اللغة. ومع ذلك، على حد علمنا، فإن الحلول الموجودة لهذا الغرض لا تستفيد استفادة كاملة من البنى الموازية الشائعة جدا في الوقت الحاضر.

هذا العمل هو خطوة أولى نحو الجمع بين التوازي مع استعمال الذاكرة الخارجية في عد (ن-غرام) نحن نبني الحل اعتماداً على مكتبة (STXXL)، والتي تتيح الوصول إلى العديد من هياكل بيانات الذاكرة الخارجية الجاهزة للاستخدام، مع دعم التوازي في مستوى الادخال/والاخراج .ثم نذهب خطوة أخرى، من خلال توزيع الحساب على العديد من الأجهزة كل منها يستخدم الحساب باستخدام انوية متعددة (threads). والنتيجة هي حل متوازٍ للغاية، والذي يمكن أن يؤدي إلى الاعتماد على مجموعات البيانات الكبيرة على أجهزة ذات موارد محدودة. الحل الذي قدمناه يمكن أن يحقق لنا ما يصل إلى 4 أضعاف السرعة على حل واحد لوحدة ذات قرص واحد.

وفقًا لمعرفتنا، المنتجات البرمجية التي تؤدي هذه المهمة (مثل KenLM) لديها القيود التالية: تقوم بافتراض أن المفردات ليست أكبر من الذاكرة الرئيسية ، ولا تدعم التوزيع المتوازي ولا تدعم نموذج القرص المتوازي.

عملنا يهدف للتغلب على حل هذه القيود عن طريق فهرسة النص بهدف منحنا القدرة على دعم مفردات كثيرة جدا، واستخدمنا (STXXL) بهدف منحنا القدرة على دعم نموذج القرص المتوازي، واستخدام عناصر مختلفة لدعم التوزيع المتوازي.

تصف هذه الرسالة العديد من التقنيات ذات الأداء الفعال المستخدم لتطبيق حلولنا.

الكلمات الرئيسية: مذكرة، معالجة_اللغات_الطبيعية، STXXL، النصوص، القرص.

Dedications

To My dear father the mercy of God, He took care of education and the reason for access to what I am now.

To my mother who has always encouraged us with her prayers, may God protect her and bring them good health and long life.

To my brother Abdelhamide who has always encouraged us and his wife Keltoume and little girl Firdousse.

To my sisters Fouzia and Fatima and Hanane and Nassira and little girl Souhila.

To my dear parents with all my feelings of respect, gratitude and gratitude for all the sacrifices made to raise me and to assure my education in the best conditions, that god gives them good health and long life.

A and all my friends and acquaintances.

I dedicate this work.

Bekraoui Med lamine

"Any accomplishment requires hard work and support, this project is no different"

Dedicated

To

My Mother

Who taught me to believe in Allah, and hard work but also to believe in myself.

My Father

Who had been a source of motivation and inspiration.

Without forgetting my little sister and brothers, you are such pure souls that can lead a man to go further and achieve more. May god bless you as well as all the family.

I also express my gratitude to Mr. "MEDIANI Mohammed" for his guidance and encouragement in carrying out this project.

I humbly thank every person who believed in me, supported and encouraged me, as I dedicate this work to all students of "2nd Years Intelligent system 2017/2018 "

SENNOUSSI FAYSSAL TAQIY EDDINE

Acknowledgements

First and foremost, we give our deep thanks to Allah for giving us the opportunity and the strength to accomplish this work.

We would like to thank our supervisor Mr.Mediani Mohammed for his help and support during our work for creating a research environment that has been very inspiring. This work would not have been possible without his guidance, encouragement and motivation. He deserves our respect and thanks.

Our respect and gratitude to the members of the jury who made us the honor of judging this work and their availability, observations and reports that have enabled us to enrich our work.

Thanks to all the teachers of the faculty of Sciences and Engineering Sciences.

I thank all my colleagues' promotion 2017/2018.

All those who participated directly or indirectly to the realization of this work.

Bekraoui Mohammed lamine** and **Sennoussi Fayssal Taqiy Eddine

Table of contents

CHAPTER 3 : IMPLEMENTATION

CHAPTER 4 CONCLUSIONS

List of Figures:

List of Table:

List of Acronyms

STXXL: Standard Template Library for Extra Large Data Sets.

NLP: Natural Language Processing

NLU: Natural Language Understanding

PDM: Parallel Disk Model

STL: Standard Template Library

CGAL: Computational Geometry Algorithms Library

TPIE: Tropical Plant International Expo

LEDA-SM: Extending LEDA to Secondary Memory

LRU: Last Recently Used

DAG: directed acyclic graph

POD: Plain Old Data type

BFS: Breadth-first search

MSD: Most Significant Digit

CHAPTER 1
INTRODUCTION

1.1. Introduction

People communicate in many different ways: through speaking and listening, making gestures, using specialised hand signals (such as when driving or directing traffic), or through various forms of text.

By text we mean words that are written or printed on a flat surface (paper, card, street signs and so on) or displayed on a screen or electronic device in order to be read by their intended recipient (or by whoever happens to be passing by). Will a computer program ever be able to convert a piece of text into a programmer friendly data structure, which describes the meaning of the natural language text? Unfortunately, no consensus has emerged about the form or the existence of such a data structure. Until such fundamental Artificial Intelligence problems are resolved, computer scientists must settle for the reduced objective of extracting simpler representations that describe limited aspects of the textual information. These simpler representations are often motivated by specific applications, or by our belief that they capture something more general about natural language. They can describe syntactic information (e.g., part-of-speech tagging, chunking, and parsing) or semantic information (e.g., word-sense disambiguation, semantic role labelling, named entity extraction, and anaphora resolution). Text corpora have been manually annotated with such data structures in order to compare the performance of various systems. The availability of standard benchmarks has stimulated research in Natural Language Processing (NLP) and effective systems have been designed for all these tasks. Such systems are often viewed as software components for constructing real-world NLP solutions.

Natural language processing is important for different reasons to different people. For some, it offers the utility of automatically harvesting arbitrary bits of knowledge from vast information resources that have only recently emerged. To others, it is a laboratory for the investigation of the human use of language--a primary cognitive ability--and its relation to thought.[1]

1.2. Natural Language Processing (NLP)

Natural Language Processing is a theoretically motivated range of computational techniques for analysing and representing naturally occurring texts at one or more levels of linguistic analysis for the purpose of achieving human-like language processing for a range of tasks or applications.[2]

The goal of NLP is “to accomplish human-like language processing”. The choice of the word ‘processing’ is very deliberate, and should not be replaced with ‘understanding’. Although the

field of NLP was originally referred to as Natural Language Understanding (NLU) in the early days of AI, it is well agreed today that while the goal of NLP is true NLU, that goal has not yet been accomplished. A full NLU System would be able to:

1. Paraphrase an input text

2. Translate the text into another language

3. Answer questions about the contents of the text

4. Draw inferences from the text

While NLP has made serious inroads into accomplishing goals 1 to 3, the fact that NLP systems cannot, of themselves, draw inferences from text, NLU still remains the goal of NLP. There are more practical goals for NLP, many related to the particular application for which it is being utilized.[3] For example, an NLP-based IR system has the goal of providing more precise, complete information in response to a user's real information need. The goal of the NLP system here is to represent the true meaning and intent of the user's query, which can be expressed as naturally in everyday language as if they were speaking to a reference librarian. Also, the contents of the documents that are being searched will be represented at all their levels of meaning so that a true match between need and response can be found, no matter how either are expressed in their surface form.[4]

In the next figure we will show you the NLP steps:

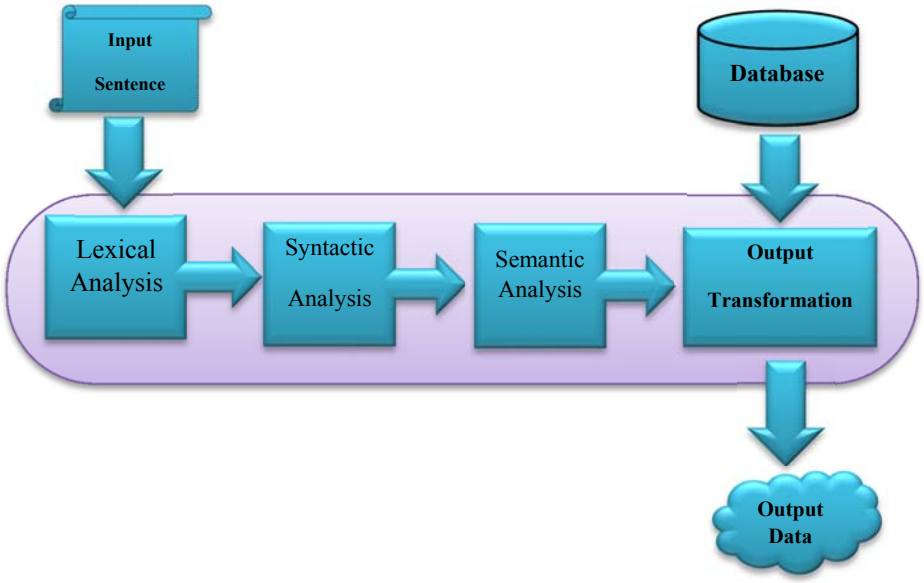

Figure 1.1 : Natural Language Processing steps

1.3. Corpora in NLP

Corpus linguistics, a branch of linguistics that deals with building corpora and investigation of their data, has already celebrated its 55th anniversary counting from the appearance of the Brown corpus. The idea of corpora that contain big data has attracted scholars' attention for a long time. During the last decade more and more corpora are being compiled automatically. From traditional text collections they vary both in their volume and content. This is closely related to the growing availability of technical resources and thus the gradually changing paradigm in corpus linguistics moving forward from "manual" approach to more automatic one. By a classical or traditional approach one can understand a compilation of corpora based on a previously described methodology: selection of texts involving their representativeness and balance, their correction, annotation and upload. New corpora contain in general texts that were automatically crawled from the Web. Researchers find it attractive to make statistical inferences on increasingly larger scope of data.[5]

The first thing to do when we got access to a new corpus is to explore the content using some basic methods, typically by counting the words. We can find out which words are the most frequent in the corpus, and by ranking the words by corpus frequency we can study the distribution of the vocabulary of the corpus. By using normalized frequencies, we can make comparisons between different corpora. We can e.g., compare the vocabulary frequency distribution of English (British National Corpus), Swedish (Stockholm-Umeå Corpus), and Swedish Sign Language (Swedish Sign Language Corpus).[6]

By using basic corpus linguistic tools, either built-in web interface tools for corpora such as COCA or BNC, or software such as AntConc, we can also look at recurring sequences of words or signs, either as sequences of tokens (called n-grams) or as collocations.[7]

Starting with basic methods such as these, we can move on to study many aspects of language production using both quantitative and qualitative methods. However, there are limitations to what corpora can tell us.

- No negative evidence: just because a word or a sign does not occur in a corpus (however large and well balanced) does not mean that the word or sign never can occur in the language. However, a representative corpus can show us what is central and typical in a language.

The findings of a study can tell us something about the subset of language that is included in that corpus, but not necessarily about language as a whole. However, if the corpus is

representative of the language we are interested in we can make careful generalizations about the language.

- A corpus can rarely provide explanations, and thus most corpus studies combine quantitative and qualitative work. Sometimes other methods, such as questionnaires, eye gaze or EEG experiments are better suited to answer a particular question. Sometimes a descriptive corpus study can give new ideas on what to look for using other methods.[8]

To summarize: make sure that you select the right corpus for your study, find out as much as you can about the corpus, take the characteristics and limitations of the corpus into account, and make careful generalizations!

Corpus analysis provides quantitative, reusable data, and an opportunity to test and challenge our ideas and intuitions about language. Further, analysis applied to corpora as transcriptions or other types of linguistic annotation can be checked for consistency and inter-annotator agreement, and the annotated corpus can be reviewed and reused by others.[9]

Corpora are essential in particular for the study of spoken and signed language: while written language can be studied by examining the text, speech, signs and gestures disappear when they have been produced and thus, we need multimodal corpora in order to study interactive face-to- face communication.[10]

The proportions suggested above relate to the characteristics of general reference corpora, and they do not necessarily hold good for other kinds of corpus. For example, it is reasonable to suppose that a corpus that is specialised within a certain subject area will have a greater concentration of vocabulary than a broad-ranging corpus, and that is certainly the case of a corpus of the English of Computing Science (James et al 1994). It is a million words in length, and some comparisons with a general corpus of the same length (the LOB corpus) are given in Table 1 (the corpus of English of Computing Science is designated as 'HK').[11]

	LOB	HK	%
Number of different word-forms (types)	69990	27210	39
Number that occur once only	36796	11430	31
Number that occur twice only	9890	3837	39
Twenty times or more	4750	3811	80
200 times or more	471	687	69

Table 1. 1 Comparison of frequencies in a general and a specialised corpus.

The number of different word forms, which is a rough estimate of the size of the vocabulary, is far less in the specialised text than it is in the general one — less than 40% of its size. The proportion of single occurrences is another indication of the spread of the vocabulary, and here the proportional difference between the two corpora is even greater, with the specialised corpus having only 31% of the total of the other corpus. Word forms which occur twice are also much less common in the specialised corpus, but the gap closes quite dramatically when we look at the figures for twenty occurrences. At a frequency of 200 and above the proportions are the other way round, and the general corpus has only 69% of the number of such words in the specialised corpus. Assuming that the distribution of the extremely common words is similar in the two corpora, these figures suggest that the specialised corpus highlights a small, probably technical vocabulary.[12]

This is only one example, but it is good news for builders of specialised corpora, in that not only are they likely to contain fewer words in all, but it seems as if the characteristic vocabulary of the special area is prominently featured in the frequency lists, and therefore that a much smaller corpus will be needed for typical studies than is needed for a general view of the language.[13]

1.4. Size of DATA

The minimum size of a corpus depends on two main factors: the kind of query that is anticipated from users, and the methodology they use to study the data.

There is no maximum size. We will begin with the kind of figures found in general reference corpora, but the principles are the same, no matter how large or small the corpus happens to be.

To relate the kind of query to the size of the corpus, it is best to start with a list of the "objects" that you intend to study; the usual objects are the physical word forms or objects created by tags, such as lemmas. Then try them out on one of the corpora that is easy to interrogate, such as the million-word corpora on the ICAME CD-ROM (Hofland 1999).[14] The Brown-group of corpora are helpful here, because they have been proof-read and tagged and edited over many years, and with a million words the sums are easy.

To illustrate how this can be done, let us take the simple case of a researcher wishing to investigate the vocabulary of a corpus. For any corpus one of the first and simplest queries is a list of word forms, which can be organised in frequency order. (NB word forms are not lemmas, where the various inflections of a "word" in the everyday sense are gathered together, but the message would not be much different with lemmas.[15]

The frequencies follow Zipf's Law (1935), which basically means that about half of them occur once only, a quarter twice only, and so on. So for the first million-word corpus of general written American English (the Brown corpus), there was a vocabulary of different word forms of 69002, of which 35065 occurred once only. At the other end of the frequency scale, the commonest word, the has a frequency of 69970, which is almost twice as common as the next one, of, at 36410.[16]

There is very little point in studying words with one occurrence, except in specialised research, for example authorship studies (Morton 1986). Recurrence — a frequency of two or more — is the minimum to establish a case for being an independent unit of the language; but only two occurrences will tell us very little indeed about the word. At this point the researcher must fix a minimum frequency below which the word form will not be the object of study. Let us suggest some outline figures that may guide practice. A word which is not specially ambiguous will require at least twenty instances for even an outline description of its behaviour to be compiled by trained lexicographers. But there are other factors to consider, the consequences of what seems to be a general point that alternatives — members of a set or system — are often not equally likely. The same tendency that we see in Zipf's Law is found in many other places in the numerical analysis of a corpus. Very often the main meaning or use or grammatical choice of a word is many times as frequent as the next one, and so on, so that twenty occurrences may be sufficient for the principal meaning of a word, while some quite familiar senses may occur only seldom. This applies also to frequent words which can have some important meanings or uses which are much less common than the principal ones. Word classes occur in very different

proportions, so if the word can be both noun and verb, the verb uses are likely to be swamped by the noun ones, and for the verb uses researchers often have recourse to a tagged corpus. In many grammatical systems one choice is nine times as common as the other, so that for every negative there are nine positives.[17]

So some additional leeway will have to be built in to cope with such contingencies. If the objects of study are lemmas rather than word forms, the picture is not very different. The minimum number of instances needed for a rough outline of usage will rise to an average of about fifty for English (but many more for highly inflected languages).[18]

If the research is about events which are more complicated than just word occurrence, then the estimate of a suitable corpus size will also get more complicated. For example if the research is about multi-word phrases, it must be remembered that the occurrence of two or more words together is inherently far rarer than either on its own. So if each of the two words in a minimal phrase occur 20 times in a million word corpus, for 20 instances of the two together the arithmetic suggests a corpus of approximately 5 billion words will be needed. For three words together of this frequency the size of the corpus could be beyond our imaginings.[19]

However, words do not occur according to the laws of chance, and if the phrases chosen are normal ones in the language, they will occur many times more often than the arithmetic projection above; so a much smaller corpus is likely to contain sufficient instances. To estimate roughly the size of a corpus for retrieval of a combination of two objects, first estimate the size you will need for the less common object on its own and then raise that figure by an order of magnitude. If there are 20 instances per million words for each of two words in a phrase, then twenty million words is likely to provide 20 instances of the pair (rather than the 5 billion projected by the arithmetic); if there are three of this frequency than 200 million words will probably be enough.[19]

These are the kinds of figures that you will need to use in estimates of your optimal corpus size. Now we must build in the considerations of the methodology that you intend to use, because this can have a dramatic effect on the size.[19]

The main methodological point is whether, having examined directly the initial results of corpus searches you intend to return to indirect methods and use the computer for further stages, recycling and refining early results6. If the latter, you will have to increase the minimum number of occurrences of your object quite substantially. This is because the regularities of occurrence that the machine will search for are not on the surface, and the way the computer

works is to examine the contexts minutely searching for frequently repeated patterns. Having found these it can then isolate instances of unusual and particular co-occurrences, which can either be discarded or studied separately after the main patterns have been described. For example, if the computer searches for the adjectives that come between in and trouble, in text sequence (Bank of English 17/10/04)[20] these are: (Unspecified, terrible, deep, serious, Cuba, serious, serious, great...) .

It is reasonable already to anticipate that deep and serious are likely to be important recurrent collocates, but single instances of the others do not offer useful evidence. In fact unspecified does not recur, terrible is a good collocate, with 33 instances out of 1729. Deep is an important collocate with 251 instances, 14.5%, while Cuba is unique, serious is slightly greater than deep at 271. Great, on the other hand, scores merely 8. The next in sequence is big, which at 235 instances is up with deep and serious. As we examine more and more instances, these three adjectives gradually separate themselves from all the others because of the number of times they appear — in total (757), almost half of all the instances. The nearest contender is real, at 142 quite considerably less common, and after that financial at 113. The computer also records as significant collocates terrible (35), dire (31) and desperate (28); deeper (14), double (14), foul (11), bad (14), such (28), enough (17) and worse (11).[20]

The pure frequency picks out the three or four collocates that are closely associated with the phrase in trouble, and reference to the statistical test (here the t-score) adds another dozen or so adjectives which, while less common in the pattern are still significantly associated and add to the general gloom that surrounds the phrase. Single occurrences like unspecified and Cuba drop into obscurity, as do terminal (2) and severe (4), which occur among the first 30 instances.

The density of the patterns of collocation is one of the determinants of the optimal size of a corpus. Other factors include the range of ambiguity of a word chosen, and sometimes its distribution among the corpus components.

If you intend to continue examining the first results using the computer, you will probably need several hundred instances of the simplest objects, so that the programs can penetrate below the surface variation and isolate the generalities. The more you can gather, the clearer and more accurate will be the picture that you get of the language.[21]

1.5. Related work

To the best of our knowledge, KenLM is the only public software which perform n-gram counting and language modelling using external memory. It was created to allow training very large language models on machines with limited memory resources. It computes the n-gram probabilities using an important number of sorting over the set of all n-grams.

KenLM defines a language model state object that speeds up the process of calculating language model probability. A state object consists of context words, back-offs and length. KenLM does use external memory for extract large corpora [22]. However, it doesn't support the parallelism, neither on the disk level nor on the processing level. Moreover, it assumes that the underlying vocabulary fits into main memory. In this work, we try to perform the counting step without these assumptions. Furthermore, we support parallelism both on the disk level and on the processing level.

1.6. External Memory For large datasets

As, mentioned before, dealing with very large corpora imposes using external memory to hold parts of the dataset while the processor is busy treating the parts kept in memory. However, in such scenario, retrieving identical data chunks (as needed in counting) would be a very expensive operation. It is a common practice to sort the large dataset (of course with the help of the external memory), so that a simple scan over the data would solve the retrieval problem.

We can't sort 1TB of data with 1GB of RAM (i.e., more data than available memory) in main memory because main memory is volatile. We want data to be saved between runs, and the data size great than the memory size that is great then address space.

The typical storage hierarchy is : CPU Registers – temporary variables, Cache – Fast copies of frequently accessed memory locations (Cache and memory should indistinguishable), Main memory (RAM) for currently used "addressable" data, and Disk for the main "big data" (secondary storage).

Solution: Utilize an External Sorting Algorithm, External sorting refers to the sorting of a file that resides on secondary memory (e.g., disk, flash, etc) ,in other side Internal sorting refers to the sorting of an array of data that is in RAM .

Objective: Minimize number of I/O accesses, external Sorting is part of the Query Evaluation, optimization subsystem and efficient Sorting algorithms can speed up query evaluation plans (e.g., during joins) [23]

1.7. External sorting

External Sorting Many important sorting applications involve processing very large files, much too large to fit into the primary memory of any computer. Methods appropriate for such applications are called external methods, since they involve a large amount of processing external to the central processing unit. There are two major factors which make external algorithms quite different:

First, the cost of accessing an item is orders of magnitude greater than any bookkeeping or calculating costs. [24]

Second, over and above with this higher cost, there are severe restrictions on access, depending on the external storage medium used: for example, items on a magnetic tape can be accessed only in a sequential manner.

The wide variety of external storage device types and costs makes the development of external sorting methods very dependent on current technology. These methods can be complicated, and many parameters affect their performance: that a clever method might go unappreciated or unused because of a simple change in the technology is a definite possibility in external sorting. For this reason, only general methods will be considered rather than specific implementations. For external sorting, the "systems" aspect of the problem is certainly as important as the "algorithms" aspect. Both areas must be carefully considered if an effective external sort is to be developed. The primary costs in external sorting are for input-output. A good exercise for someone planning to implement an efficient program to sort a very large file is first to implement an efficient program to copy a large file, then (if that was too easy) implement an efficient program to reverse the order of the elements in a large file. The systems problems that arise in trying to solve these problems efficiently are similar to those that arise in external sorts. Permuting a large external file in any non-trivial way is about as difficult as sorting it, even though no key comparisons, etc. are required. In external sorting, we are concerned mainly with limiting the number of times each piece of data is moved between the external storage medium and the primary memory, and being sure that such transfers are done as efficiently as allowed by the available hardware. External sorting methods have been developed which are suitable

for the punched cards and paper tape of the past, the magnetic tapes and disks of the present, and emerging technologies such as bubble memories and videodisks. The essential differences among the various devices are the relative size and speed of available storage and the types of data access restrictions. We'll concentrate on basic methods for sorting on magnetic tape and disk because these devices are in widespread use and illustrate the two fundamentally different modes of access that characterize many external storage systems. Often, modern computer systems have a "storage hierarchy" of several progressively slower, cheaper, and larger memories. Many of the algorithms can be adapted to run well in such an environment, but we'll deal exclusively with "two-level" memory hierarchies consisting of main memory and disk or tape.[25]

1.8. Conclusions

This chapter describes an example of integration between big data and text analysis techniques that can provide inspiration for future research. The possibility of quickly extracting a (well-defined) selection of tweets and computing variables associated to them is a promising starting point for research.

Analysis techniques that do not require language resources, such as repeated segments detection and extraction of characteristic words, can be applied more straightforwardly as they do not pose limitations on the language to select. Among the resources that are more specific to text analysis the use of the positive-negative dictionary included in Taltac2 was particularly useful for classifying the mood of the text, especially in the presence of events of high media coverage and emotional impact.

The recognition of hashtags can be considered an added value also in the analysis of content and user profiles, for example analysing the number of “#” in a text or the way they are used (attached to words in the text or as separated tags). The same can be said about the use of other special characters, such as “@” for targeting other users or links to images and web pages.

The results of the analysis are unavoidably influenced by the use of a geographical filter on the source. However, the concentration of major media and political influencers makes the selected sample highly representative for what concerns the different types of users in an explorative analysis. Nevertheless, future work could explore the effect of different or wider geographical selections of publication place. [26]

CHAPTER 2
STXXL

2.1 INTRODUCTION

Massive data sets arise naturally in many domains. Spatial data bases of geographic information systems like Google Earth and NASA's World Wind store terabytes of geographically-referenced information that includes the whole Earth. In computer graphics one has to visualize highly complex scenes using only a conventional workstation with limited memory. Billing systems of telecommunication companies evaluate terabytes of phone call log files. One is interested in analysing huge network instances like a web graph or a phone call graph. Search engines like Google and Yahoo provide fast text search in their data bases indexing billions of web pages. A precise simulation of the Earth's climate needs to manipulate with petabytes of data. These examples are only a sample of numerous applications which have to process vast amounts of data. [27]

The internal memories of computers can keep only a small fraction of these large data sets. During the processing the applications need to access the external memory (e. g. hard disks) very frequently. One such access can be about 106 times slower than a main memory access. Therefore, the disk accesses (I/Os) become the main bottleneck.

The data is stored on the magnetic surface of a hard disk that rotates 4200– 15000 times per minute. In order to read or write a designated track of data, the disk controller moves the read/write arm to the position of this track (seek latency). If only a part of the track is needed, there is an additional rotational delay. The total time for such a disk access is an average of 3–10 ms for modern disks. The latency depends on the size and rotational speed of the disk and can hardly be reduced because of the mechanical nature of hard disk technology. After placing the read/write arm, the data is streamed at a high speed which is limited only by the surface data density and the bandwidth of the I/O interface. This speed is called sustained throughput and achieves up to 80 MByte/s nowadays. In order to amortize the high seek latency, one reads or writes the data in blocks. The block size is balanced when the seek latency is a fraction of the sustained transfer time for the block. Good results show blocks containing a full track. For older low density disks of the early 90's the track capacities were about 16-64 KB. Nowadays, disk tracks have a capacity of several megabytes.

Operating systems implement the virtual memory mechanism that extends the working space for applications, mapping an external memory file (page/swap file) to virtual addresses. This idea supports the Random Access Machine model in which a program has an infinitely large main memory. With virtual memory the application does not know where its data is located: in

the main memory or in the swap file. This abstraction does not have large running time penalties for simple sequential access patterns: The operating system is even able to predict them and to load the data in ahead. For more complicated patterns these remedies are not useful and even counterproductive: the swap file is accessed very frequently; the executable code can be swapped out in favour of unnecessary data; the swap file is highly fragmented and thus many random I/O operations are needed even for scanning. [27]

2.2 I/O-efficient algorithms and models

The operating system cannot adapt to complicated access patterns of applications dealing with massive data sets. Therefore, there is a need of explicit handling of external memory accesses. The applications and their underlying algorithms and data structures should care about the pattern and the number of external memory accesses (I/Os) which they cause.

Several simple models have been introduced for designing I/O-efficient algorithms and data structures (also called external memory algorithms and data structures). The most popular and realistic model is the Parallel Disk Model (PDM) of Vitter and Shriver. In this model, I/Os are handled explicitly by the application. An I/O operation transfers a block of B consecutive elements from/to a disk to amortize the latency. [28]

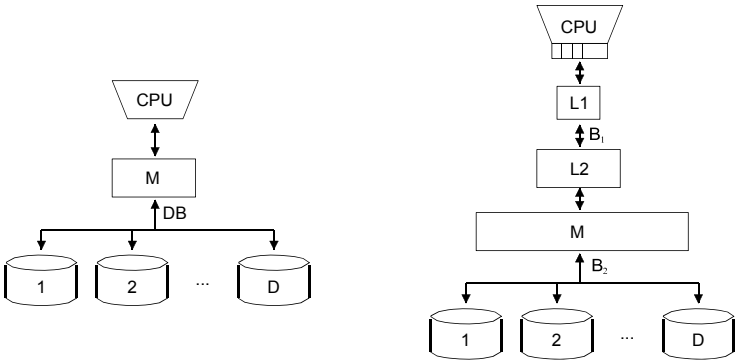

Figure 2.1 Schemes of parallel disk model (left) and memory hierarchy (right).

The application tries to transfer D blocks between the main memory of size M bytes and D independent disks in one I/O step to improve bandwidth, see Figure 1. The input size is N bytes which is (much) larger than M. The main complexity metrics of an I/O-efficient algorithm in PDM are the number of I/O steps (main metric) and the number of operations executed by the

CPU. If not I/O but a slow internal CPU processing is the limiting factor of the performance of an application, we call such behaviour CPU-bound.

The PDM measures the transfers between the main memory and the hard disks, however, in modern architectures, the CPU does not access the main memory directly. There are a few levels of faster memory caches in-between (Figure 1): CPU registers, level one (L2), level two (L2) and even level three (L3) caches. The main memory is cheaper and slower than the caches. Cheap dynamic random access memory, used in the majority of computer systems, has an access latency up to 60 ns whereas L1 has a latency of less than a ns. However, for a streamed access a high bandwidth of several GByte/s can be achieved. The discrepancy between the speed of CPUs and the latency of the lower hierarchy levels grows very quickly: the speed of processors is improved by about 55 % yearly, the hard disk access latency only by 9 % . Therefore, the algorithms which are aware of the memory hierarchy will continue to benefit in the future and the development of such algorithms is an important trend in computer science. The PDM model only describes a single level in the hierarchy. An algorithm tuned to make a minimum number of I/Os between two particular levels could be I/O-inefficient on other levels.

The cache-oblivious model in avoids this problem by not providing the knowledge of the block size B and main memory size M to the algorithm. The benefit of such an algorithm is that it is I/O-efficient on all levels of the memory hierarchy across many systems without fine tuning for any particular real machine parameters. Many basic algorithms and data structures have been designed for this model. A drawback of cache-oblivious algorithms playing a role in practice is that they are only asymptotically I/O-optimal. The constants hidden in the O-notation of their I/O-complexity are significantly larger than the constants of the corresponding I/O-efficient PDM algorithms (on a particular memory hierarchy level). For instance, a tuned cache-oblivious funnel sort implementation is 2.6–4.0 times slower than our I/O-efficient sorter from STXXL for out-of-memory inputs . A similar funnel sort implementation is up to two times slower than the I/O-efficient sorter from the TPIE library for large inputs. The reason for this is that these I/O-efficient sorters are highly optimized to minimize the number of transfers between the main memory and the hard disks where the imbalance in the access latency is the largest. Cache-oblivious implementations tend to lose on the inputs, exceeding the main memory size, because they do (a constant factor) more I/Os at the last level of memory hierarchy. In this paper we concentrate on extremely large out-of- memory inputs, therefore we will design and implement algorithms and data structures efficient in the PDM. [28]

2.2.1 Algorithm engineering for large data sets

Theoretically, I/O-efficient algorithms and data structures have been developed for many problem domains: graph algorithms, string processing, computational geometry, etc. (see the surveys). Some of them have been implemented: sorting, matrix multiplication, search trees, priority queues , text processing . However only few of the existing I/O-efficient algorithms have been studied experimentally. As new algorithmic results rely on previous ones, researchers, which would like to engineer practical implementations of their ideas and show the feasibility of external memory computation for the solved problem, need to invest much time in the careful design of unimplemented underlying external algorithms and data structures. Additionally, since I/O-efficient algorithms deal with hard disks, a good knowledge of low-level operating system issues is required when implementing details of I/O accesses and file system management. This delays the transfer of theoretical results into practical applications, which will have a tangible impact for industry. Therefore one of the primary goals of algorithm engineering for large data sets is to create software frameworks and libraries which handle both the low-level I/O details efficiently and in an abstract way, and provide well-engineered and robust implementations of basic external memory algorithms and data structures.[27]

2.2.2 C++ standard template library

The Standard Template Library (STL) is a C++ library which is included in every C++ compiler distribution. It provides basic data structures (called containers) and algorithms. STL containers are generic and can store any built-in or user data type that supports some elementary operations (e.g. copying and assignment). STL algorithms are not bound to a particular container: an algorithm can be applied to any container that supports the operations required for this algorithm (e.g. random access to its elements). This flexibility significantly reduces the complexity of the library.

STL is based on the C++ template mechanism. The flexibility is supported using compile-time polymorphism rather than the object oriented run-time polymorphism. The run-time polymorphism is implemented in languages like C++ with the help of virtual functions that usually cannot be inlined by C++ compilers. This results in a high per-element penalty of calling a virtual function. In contrast, modern C++ compilers minimize the abstraction penalty of STL inlining many functions.

STL containers include: std::vector (an unbounded array), std::priority queue, std::list, std::stack, std::deque, std::set, std::multiset (allows duplicate elements), std::map (allows mapping from one data item (a key) to another (a value)), std::multimap (allows duplicate keys), etc. Containers based on hashing (hash set, hash multiset, hash map and hash multimap) are not yet standardized and distributed as an STL extension.

Iterators are an important part of the STL library. An iterator is a kind of handle used to access items stored in data structures. Iterators offer the following operations: read/write the value pointed by the iterator, move to the next/previous element in the container, move by some number of elements forward/backward (random access). [28]

STL provides a large number of algorithms that perform scanning, searching and sorting. The implementations accept iterators that posses a certain set of operations described above. Thus, the STL algorithms will work on any container with iterators following the requirements. To achieve flexibility, STL algorithms are parameterized with objects, overloading the function operator (operator()). Such objects are called functors. A functor can, for instance, define the sorting order for the STL sorting algorithm or keep the state information in functions passed to other functions. Since the type of the functor is a template parameter of an STL algorithm, the function operator does not need to be virtual and can easily be inlined by the compiler, thus avoiding the function call costs.

The STL library is well accepted and its generic approach and principles are followed in other famous C++ libraries like Boost and CGAL. [28]

2.2.3 The goals of STXXL

Several external memory software library projects (LEDA-SM and TPIE) were started to reduce the gap between theory and practice in external memory computing. They offer frameworks which aim to speed up the process of implementing I/O-efficient algorithms, abstracting away the details of how I/O is performed.

The motivation for another project, namely STXXL, was that an easier to use and higher performance library was needed. Here are a number of key new or improved features of STXXL:

- Transparent support of parallel disks. The library provides implementations of basic parallel disk algorithms. STXXL is the only external memory algorithm library supporting parallel disks. Such a feature was announced for TPIE in 1996.

- The library is able to handle problems of a very large size (up to dozens of terabytes).
- Improved utilization of computer resources. STXXL explicit supports overlapping between I/O and computation. STXXL implementations of external memory algorithms and data structures benefit from the overlapping of I/O and computation.
- STXXL achieves small constant factors in I/O volume. In particular, “pipelining” can save more than half the number of I/Os performed by many algorithms.
- Short development times due to well-known STL-compatible interfaces for external memory algorithms and data structures. STL algorithms can be directly applied to STXXL containers (code reuse); moreover, the I/O complexity of the algorithms remains optimal in most cases. [27]

2.2.4 Overview

The chapter is structured as follows. Section 3 overviews the design of STXXL. We explain the design decisions we have made to achieve high performance in Sections 4–6 in detail. Section 7 engineers an efficient parallel disk sorting, which is the most important component of an external memory library. The concept of algorithm pipelining is described in Section 8. The design of our implementation of pipelining is presented in Section 9. Section 10 gives a short overview of the projects using STXXL. We make some concluding remarks and point out the directions of future work in Section 11.

The shortened preliminary material of this paper has been published in conference proceedings.

2.3 THE DESIGN OF THE STXXL LIBRARY

STXXL is a layered library consisting of three layers (see Figure 2). The lowest layer, the Asynchronous I/O primitives layer (AIO layer), abstracts away the details of how asynchronous I/O is performed on a particular operating system. Other existing external memory algorithm libraries only rely on synchronous I/O APIs or allow reading ahead sequences stored in a file using the POSIX asynchronous I/O API . These libraries also rely on uncontrolled operating system I/O caching and buffering in order to overlap I/O and computation in some way. However, this approach has significant performance penalties for accesses without locality. Unfortunately, the asynchronous I/O APIs are very different for different operating systems (e.g. POSIX AIO and Win32 Overlapped I/O). Therefore, we have introduced the AIO layer was introduces to make porting STXXL easy. Porting the whole library to a different platform

requires only implementing the thin AIO layer using native file access methods and/or native multithreading mechanisms.

The Block Management layer (BM layer) provides a programming interface emulating the parallel disk model. The BM layer provides an abstraction for a fundamental concept in the external memory algorithm design — a block of elements. The block manager implements block allocation/deallocation, allowing several block-to-disk assignment strategies: striping, randomized striping, randomized cycling, etc. The block management layer provides an implementation of parallel disk buffered writing, optimal prefetching, and block caching. The implementations are fully asynchronous and designed to explicitly support overlapping between I/O and computation.

The top of STXXL consists of two modules. The STL-user layer provides external memory sorting, external memory stack, external memory priority queue, etc. which have (almost) the same interfaces. [27]

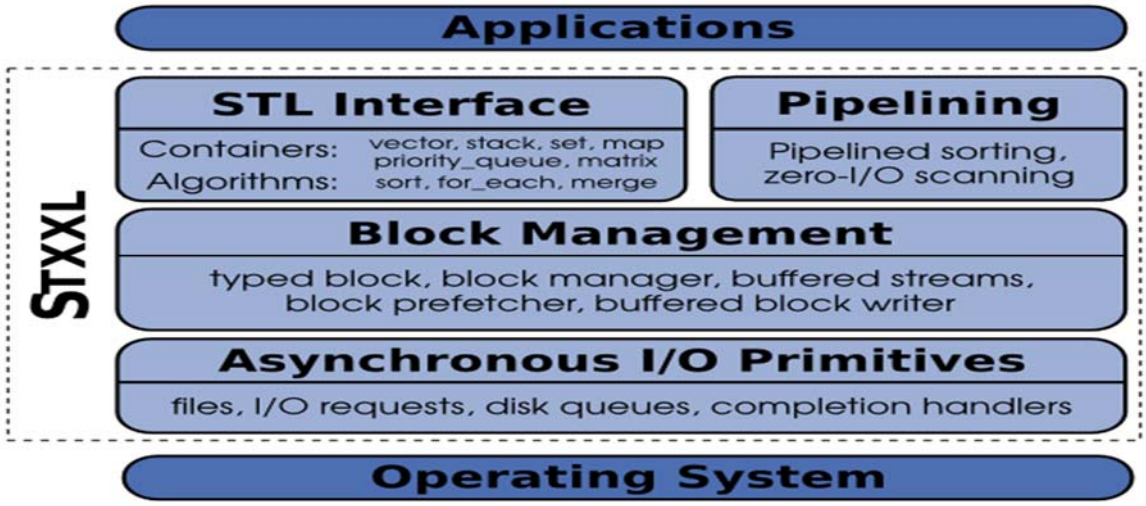

Figure 2.2. Structure of STXXL

(including syntax and semantics) as their STL counterparts. The Streaming layer provides efficient support for pipelining external memory algorithms. Many external memory algorithms, implemented using this layer, can save a factor of 2–3 in I/Os. For example, the algorithms for external memory suffix array construction implemented with this module require only 1/3 of the number of I/Os which must be performed by implementations that use conventional data structures and algorithms (either from the STXXL STL-user layer, LEDA-SM, or TPIE). The win is due to an efficient interface that couples the input and the output of

the algorithm–components (scans, sorts, etc.). The output from an algorithm is directly fed into another algorithm as input, without needing to store it on the disk in-between. This generic pipelining interface is the first of this kind for external memory algorithms.

2.4 AIO LAYER

The purpose of the AIO layer is to provide a unified approach to asynchronous I/O. The layer hides details of native asynchronous I/O interfaces of an operating system. Studying the patterns of I/O accesses of external memory algorithms and data structures, we have identified the following functionality that should be provided by the AIO layer:

- To issue read and write requests without having to wait for them to be completed.
- To wait for the completion of a subset of issued I/O requests.
- To wait for the completion of at least one request from a subset of issued I/O requests.
- To poll the completion status of any I/O request.
- To assign a callback function to an I/O request which is called upon I/O completion (asynchronous notification of completion status), with the ability to co-relate callback events with the issued I/O requests. [28]

2.5 BM LAYER

The BM layer includes a toolbox for allocating, deallocating, buffered writing, prefetching, and caching of blocks. The external memory manager (object block manager) is responsible for allocating and deallocating external memory space on disks. The manager supports four parallel disk allocation strategies: simple striping, fully randomized, simple randomized, and randomized cycling.

2.6 STL-USER LAYER

When we started to develop the library we decided to equip our implementations of external memory data structures and algorithms with well known generic interfaces of the Standard Template Library, which is a part of the C++ standard. This choice would shorten the application development times, since the time to learn new interfaces is saved. Porting an internal memory code that relies on STL would also be easy, since interfaces of STL-user layer

data structures (containers in the STL terminology) and algorithms have the same syntax and semantics. We go over the containers currently available in STXXL. [27]

2.6.1 Vector

STL vector is an array, supporting random access to elements, constant time insertion and removal of elements at the end. The size of a vector may vary dynamically. The implementation of stxxl::vector is similar to the LEDA-SM array. The content of a vector is striped block wise over the disks, using an assignment strategy given as a template parameter. Some of the blocks are cached in a vector cache of fixed size (also a parameter). The replacement of cache blocks is controlled by a specified page-replacement strategy. STXXL has implementations of LRU and random replacement strategies. The user can provide his/her own strategy as well. The STXXL vector has STL-compatible Random Access Iterators. One random access costs O(1) I/Os in the worst case. Sequential scanning of the vector costs O(1/DB) amortized I/Os per vector element.

2.6.2 Stack

A stack is a LIFO data structure that provides insertion, removal, and inspection of the element at the top of the stack. Due to the restricted set of operations a stack can be implemented I/O-efficiently and applied in many external memory algorithms. Four implementations of a stack are available in STXXL, which are optimized for different access patterns (long or short random insert/remove sequences) and manage their memory space differently (own or shared block pools). Some of the implementations (e.g. stxxl::grow shrink stack2) are optimized to prefetch data ahead and to queue writing, efficiently overlapping I/O and computation. The amortized I/O complexity for push and pop stack operations is O (1/DB). [28]

2.6.3 Queue and Deque

The design STXXL FIFO queue of is similar to stxxl::grow shrink stack2. The implementation holds the head and the tail blocks in the main memory. Prefetch and write block pools are used to overlap I/O and computation during queue operations.

The STXXL implementation of external memory deque is an adaptor of an (external memory) vector. This implementation wraps the elements around the end of the vector circularly. It provides the pop/push operations from/to both ends of the deque in O(1/DB) amortized I/Os if parameterized with a properly configured stxxl::vector.

2.6.4 Priority queue

External memory priority queues are the central data structures for many I/O efficient graph algorithms. The main technique in these algorithms is time-forward processing, easily realizable by an I/O efficient priority queue. This approach evaluates a DAG with labelled nodes.

2.6.5 Map

The map is an STL interface for search trees with unique keys. Implementation of map is a variant of a B+-tree data structure supporting the operations insert, erase, find, lower bound and upper bound. Operations of map use iterators to refer to the elements stored in the container, e.g. find and insert return an iterator pointing to the data.

2.6.6 General issues concerning STXXL containers

Similar to other external memory algorithm libraries, STXXL has the restriction that the data types stored in the containers cannot have C/C++ pointers or references to other elements of external memory containers. The reason is that these pointers and references get invalidated when the blocks containing the elements they point/refer to are written to disk. To get around this problem, the links can be kept in the form of external memory iterators (e.g. stxxl::vector::iterator). The iterators remain valid while storing to and loading from the external memory. When dereferencing an external memory iterator, the referenced object is loaded from external memory by the library on demand (if the object is not in the cache of the data structure already). [27]

STXXL containers differ from STL containers in treating allocation and distinguishing between uninitialized and initialized memory. STXXL containers assume that the data types they store are plain old data types (POD). The constructors and destructors of the contained data types are not called when a container changes its size. The support of constructors and destructors would imply a significant I/O cost penalty, e.g. on the deallocation of a non-empty container, one has to load all contained objects and call their destructors. This restriction sounds more severe than it is, since external memory data structures cannot cope with custom dynamic memory management anyway, which is the common use of custom constructors/destructors.

2.6.7 Algorithms

The algorithms of the STL can be divided into two groups by their memory access pattern: scanning algorithms and random access algorithms.

Scanning algorithms:

Scanning algorithms work with Input, Output, Forward, and Bidirectional iterators only. Since random access operations are not allowed with these kinds of iterators, the algorithms inherently exhibit a strong spatial locality of reference. STXXL containers and their iterators are STL-compatible, therefore one can directly apply STL scanning algorithms to them, and they will run I/O-efficiently. Scanning algorithms are the majority of the STL algorithms. STXXL also offers specialized implementations of some scanning algorithms (stxxl::for each, stxxl::generate, etc.), which perform better in terms of constant factors in the I/O volume and internal CPU work. These implementations benefit from accessing lower level interfaces of the BM layer instead of using iterator interfaces, resulting in a smaller CPU overhead. Being aware of the sequential access pattern of the applied algorithm, the STXXL implementations can do prefetching and use queued writing, thereby leading to the overlapping of I/O with computation.

2.7 PARALLEL DISK SORTING

Sorting is the first component we have designed for STXXL, because it is the fundamental tool for I/O- efficient processing of large data sets. Therefore, an efficient implementation of sorting largely defines the performance of an external memory software library as a whole. To achieve the best performance our implementation [29] uses parallel disks,

No previous implementation has all these properties, which are needed for a good practical sorting. LEDA-SM and TPIE concentrate on single disk implementations. For the overlapping of I/O and computation they rely on prefetching and caching provided by the operating system, which is suboptimal since it does not use prefetching information.

Barve and Vitter implemented a parallel disk algorithm [30] that can be viewed as the immediate ancestor of our algorithm. Innovations with respect to our sorting are: a different allocation strategy that enables better theoretical I/O bounds; a prefetching algorithm that optimizes the number of I/O steps and never evicts data previously fetched; overlapping of I/O and computation; a completely asynchronous implementation that reacts flexibly to fluctuations in disk speeds; and an implementation that sorts many GBytes and does not have to limit

internal memory size artificially to obtain a nontrivial number of runs. Additionally, our implementation is not a prototype, it has a generic interface and is a part of the software library STXXL.

Algorithms in [31] have the theoretical advantage of being deterministic. However, they need three passes over data even for relatively small inputs.

Prefetch buffers for disk load balancing and overlapping of I/O and computation have been intensively studied for external memory merge sort. But we have not seen results that guarantee overlapping of I/O and computation during the parallel disk merging of arbitrary runs.

2.7.1 Implementation details

Runs are build of a size close to M/2 but there are some differences to the simple algorithm. Overlapping of I/O and computation is achieved using the call-back mechanism supported by the I/O layer. Thus, the sorter remains portable over different operating systems with different interfaces to threading.

We have two implementations with respect to the internal work: stxxl::sort is a comparison based sorting using std::sort from STL to sort the runs internally;

stxxl::ksort exploits integer keys and has smaller internal memory bandwidth requirements for large elements with small key fields. After reading elements, we extract pairs (key, pointerToElement), sort these pairs, and only then move elements in sorted order to write buffers from where they are output. For reading and writing we have used unbuffered direct file I/O.

Furthermore, we exploit random keys. We use two passes of MSD (most significant digit) radix sort of the key-pointer pairs. The first pass uses the m most significant bits where m is a tuning parameter depending on the size of the processor caches and of the TLB (translation look-aside buffer). This pass consists of a counting phase that determines bucket sizes and a distribution phase that moves pairs. The counting phase is fused into a single loop with pair extraction. The second pass of radix sort uses a number of bits that brings us closest to an expected bucket size of two. This two-pass algorithm is much more cache efficient than a one-pass radix sort. The remaining buckets are sorted using a comparison based algorithm: Optimal straight line code for $n \leq 4$, insertion sort for $n \in \{5..16\}$, and quicksort for $n > 16$. Multi-way Merging. We have adapted the tuned multi-way merger from , i.e. a tournament tree stores pointers to the current elements of each merge buffer.

Overlapping I/O and Computation. We integrate the prefetch buffer and the overlap buffer to a read buffer. We distribute the buffer space between the two purposes of minimizing disk idle time and overlapping I/O and computation indirectly by computing an optimal prefetch sequence for a smaller buffer space.

Asynchronous I/O. I/O is performed without any synchronization between the disks. The prefetcher computes a sequence σ of blocks indicating the order in which blocks should be fetched. As soon as a buffer block becomes available for prefetching, it is used to generate an asynchronous read request for the next block in σ. The AIO layer queues this request at the disk storing the block to be fetched. The thread for this disk serves the queued request in FIFO manner. All I/O is implemented without superfluous copying. Blocks, fetched using unbuffered direct I/O, travel to the prefetch/overlap buffer and from there to a merge buffer simply by passing pointers to blocks. Similarly, when an element is merged, it is directly moved from the merge buffer to the write buffer and a block of the write buffer is passed to the output queue of a disk simply by passing a block pointer to the AIO layer that then uses unbuffered direct I/O to output the data.

2.7.2 Discussion

A sorting algorithm has been engineered that combines a very high performance on state of the art hardware with theoretical performance guarantees. This algorithm is compute-bound although we use small random keys and a tuned linear time algorithm for the run formation. Similar observations apply to other external memory algorithms that exhibit a good spatial locality, i.e. those dominated by scanning, sorting, and similar operations. This indicates that bandwidth is no longer a limiting factor for most external memory algorithms if parallel disks are used.

2.8 ALGORITHM PIPELINING

The pipelined processing technique is very well known in the database world.

Usually, the interface of an external memory algorithm assumes that it reads the input from (an) external memory container(s) and writes output into (an) external memory container(s). The idea of pipelining is to equip the external memory algorithms with a new interface that allows them to feed the output as a data stream directly to the algorithm that consumes the output, rather than writing it to the external memory first. Logically, the input of an external memory

algorithm does not have to reside in the external memory, rather, it could be a data stream produced by another external memory algorithm.

Many external memory algorithms can be viewed as a data flow through a directed acyclic graph G with node set V = F ∪ S ∪ R and edge set E . The file nodes F represent physical data sources and data sinks, which are stored on disks (e.g. in the external memory containers of the STL-user

layer). A file node writes or/and reads one stream of elements. The streaming nodes S read zero, one or several streams and output zero, one or several new streams. Streaming nodes are equivalent to scan operations in non-pipelined external memory algorithms. The difference is that non-pipelined conventional scanning needs a linear number of I/Os, whereas streaming nodes usually do not perform any I/O, unless a node needs to access external memory data structures (stacks, priority queues, etc.). The sorting nodes R read a stream and output it in a sorted order. Edges E in the graph G denote the directions of data flow between nodes.

2.9 STREAMING LAYER

The streaming layer provides a framework for the pipelined processing of large sequences. To the best of our knowledge we are the first who apply the pipelining method systematically in the domain of external memory algorithms. We introduce it in the context of an external memory software library.

In STXXL, all data flow node implementations have an STXXL stream interface which is similar to the STL Input iterators. As an input iterator, an STXXL stream object may be dereferenced to refer to some object and may be incremented to proceed to the next object in the stream. The reference obtained by dereferencing is read-only and must be convertible to the value type of the STXXL stream. The concept of the STXXL stream also defines a Boolean member function empty () which returns true if the end of the stream is reached.

2.10 STXXL APPLICATIONS

The STXXL library already has users both in academia and industry. We know of at least 17 institutions which apply the library for a wide range of problems including text processing, graph algorithms , gaussian elimination [32], visualization and analysis of 3D and 4D microscopic images, differential cryptographic analysis, computational geometry [33], topology analysis of large networks, statistics and time series analysis, and analysis of seismic files.

STXXL has been successfully applied in implementation projects that studied various I/O-efficient algorithms from the practical point of view. The fast algorithmic components of STXXL library gave the implementations an opportunity to solve problems of very large size on a low-cost hardware in a record time. For the tests many real-world and synthetic inputs have been used. It has been shown that external memory computation for these problems is practically feasible now. We overview some computational results of these projects.

The performance of external memory suffix array construction algorithms was investigated in . The experimentation with pipelined STXXL implementations of the algorithms has shown that computing suffix arrays in external memory is feasible even on a low-cost machine. Suffix arrays for long strings up to 4 billion characters could be computed in hours.

The project [34] has compared experimentally two external memory breadth-first search (BFS) algorithms. The pipelining technique of STXXL has helped to save a factor of 2–3 in I/O volume of the BFS implementations. Using STXXL, it became possible to compute BFS decomposition of node-set of large grid graphs with 128 million edges in less than a day, and for random sparse graph class within an hour. Recently, the results have been significantly improved.

2.11 CONCLUSION

We have described STXXL: a library for external memory computation that aims for high performance and ease-of-use. The library supports parallel disks and explicitly overlaps I/O and computation. The library is easy to use for people who know the C++ Standard Template Library. The library implementations outperform or can at least compete with the best available practical implementations on real and random inputs. STXXL supports algorithm pipelining, which saves many I/Os for many external memory algorithms.

CHAPTER 3
IMPLEMENTATION

3.1 Implementation detail

In our project we use different implementation to reach our objective in this work, this what we going to introduce in this chapter by presenting the different steps of a complete system of distributed memory-bound word counting for large data and the working Environment by showing our experimental result to show the efficacy of our software which was implemented with the programming language C++, because it made it is easy for external memory implementation.

This chapter will give u the reality of our word counting system and evaluate it performance, which was implemented according to a external memory method and the use of STXXL of the external memory computing

3.2 External Data structures

The main motivation behind using external data structure is to allow our software to handle very large input data. We perform the counting over this large input data through the following steps: Vocabulary indexing, Vector populating, Vector sorting and counting.

3.2.1 Vocabulary indexing:

STXXL, like other external memory implementations, only supports the so-called Plain Old Data (POD) structures. These are fixed size simple types such as numbers and characters as well as fixed-size structures. This constraint is traded for more efficient IO performance.

Unfortunately, our main data representations, strings, do not fit into the POD category, as they tend to be of variable lengths. Therefore, finding a mechanism to encode our data into fixed sized structures is mandatory. At the time of designing our software, we thought of two solutions:

1. Use fixed size character strings: Here we need to assume a maximum length of the input words. In other words, we accept words up to a certain length and truncate the very long ones or simply ignore them. While this assumption is reasonable for most NLP corpora, we found that it could slightly limit the flexibility of our final product. More importantly, it could lead to wasting non negligible memory amounts, when most n-grams are shorter than the assumed maximum.
2. Index our vocabulary and use the index instead of the words. With this, we have more advantages compared to using fixed size strings. Moreover, Integers are fast to create

and can be placed right in the stack, whereas strings must be in the heap. An additional advantage would be a reduced memory usage, as we expect the average word length to be greater than the size of the integer used as index.

So we chose the second solution over the first one.

Our way to implement the word indexing is to use a map which is an associative data structure and its mainly used for fast lookups or searching data. It stores data in the form of key and value pairs where every key is unique.

Nevertheless, using the map types offered by the C++ STL library might sound controversial to our original aims, as these reside fully in memory. Again, to support extremely large vocabularies we went for the map type offered by the Berkeley DB API. It is implemented using B-Trees.

We did this using Oracle Berkeley DB, because it stores data quickly and easily without the overhead found in other databases, and it supports large data volumes. Berkeley DB is a C library that runs in the same process as our application, avoiding the interposes communication delays of using a remote database server. Shared caches keep the most active data in memory, avoiding costly disk accesses.

Every processing unit should have the same and exact copy of this map. This is achieved by forcing the indexing operation in each node regardless whether this unit will treat the underlying sentence or not (the load is distributed over the computing units on a sentence basis, as will be explained in the next section) see the next figure.

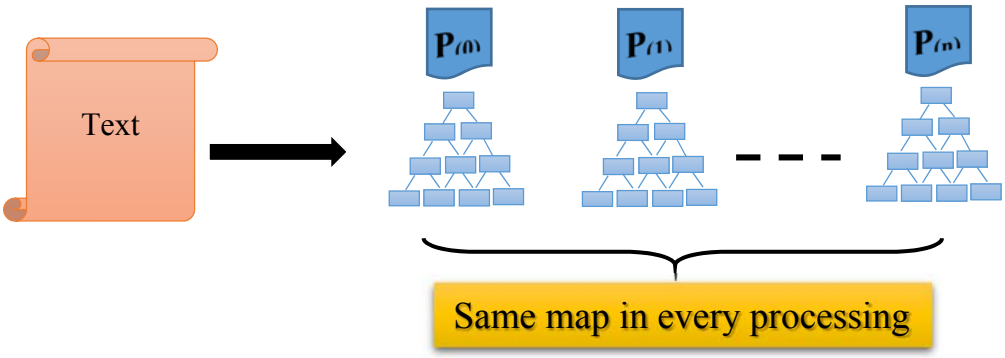

Figure 3.1: indexing process

3.2.2 Vector populating

After having the index we searched for solution that give to know the word a index , because knowing that with the map will take lot of time .

To solve this problem with a vector , that we have fill it with word in order of their index it will give us the possibility to know the word from the index directly ,see figure 3.2 .for that we use DB Vectors it a 'Hash' this is built into Berkeley-DB . vector are better and convenient way of storing the data of same data type with same size , allows us to store known number of elements in it this 2 fact are helpful for our large data.

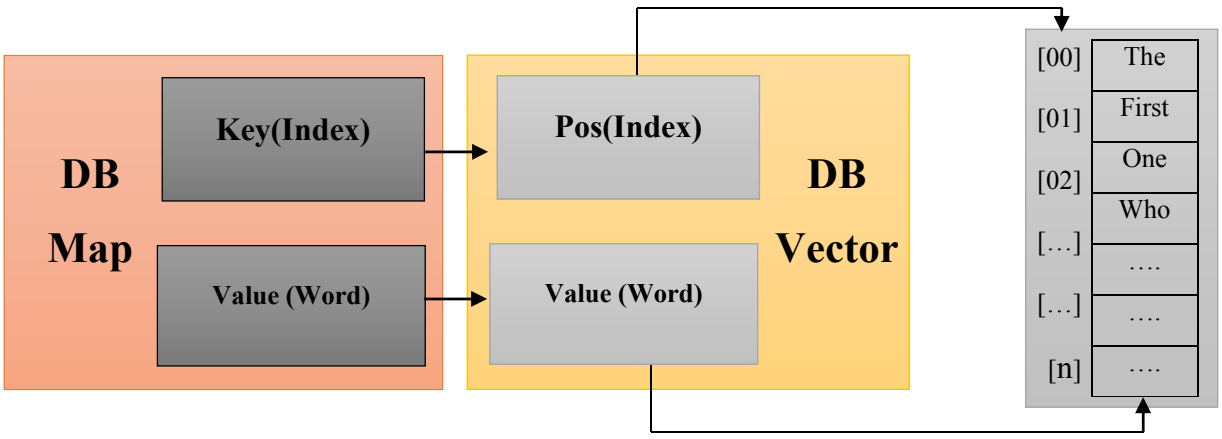

Figure 3.2: Berkeley DB structures used in the application and their interaction

After having the index of words we store n-gram of "index-word" in an STXXL vector. Unlike the vocabulary index (described in the previous step), these STXXL vectors are distributed over the computing units; and not necessarily contain identical data it shown in the next figure.

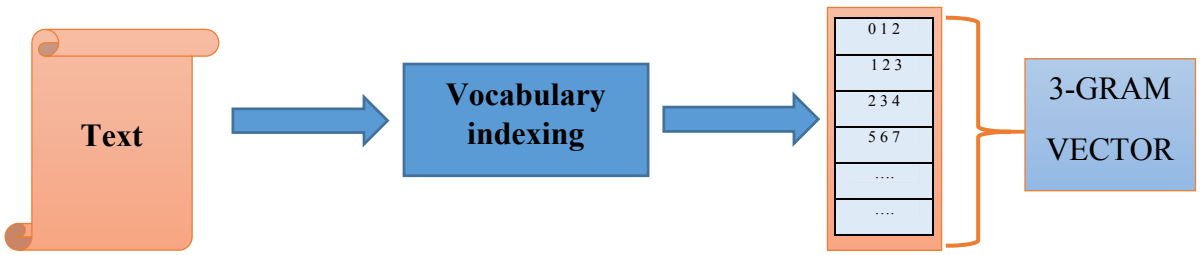

Figure 3.3: Presentation of the N-grams STXXL VECTOR

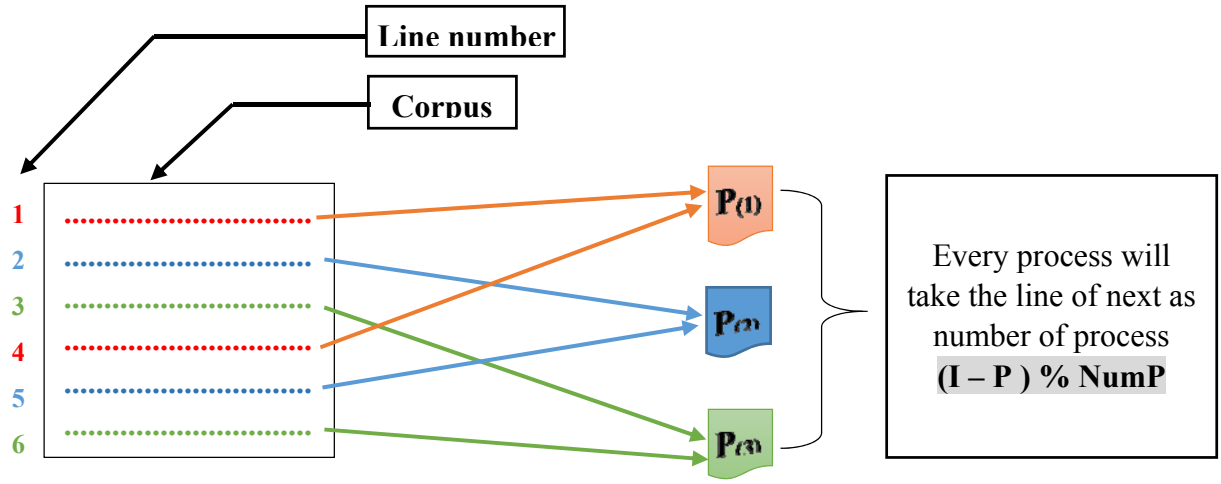

Figure 3.4: this figure show how every process fill the n-gram vectors

3.2.3 Vector sorting

Before proceeding into counting the n-grams, we need to put identical ones next to each other in the containing vector. This is obtained by sorting the STXXL vector. This operation involves distributed external sorting as the underlying container is itself distributed over multiple units.

This is the most what our work is based, the objective is to get external sorting with see figure3.5.

The most universal STXXL container is **stxxl::vector**. Vector is an array whose size can vary dynamically. The implementation of **stxxl::vector** is similar to the LEDA-SM array **[35]**. The content of a vector is striped block-wise over the disks, using an assignment strategy given as a template parameter. Some of the blocks are cached in a vector cache of fixed size (also a parameter). The replacement of cache blocks is controlled by a specified page-replacement strategy. STXXL has implementations of LRU and random replacement strategies. The user can provide his/her own strategy as well. The **stxxl::vector** has STL-compatible Random Access Iterators.

The motivation of using of the STXXL library was:

Low CPU overhead (use low-level optimizations like copying operands in CPU registers, unroll loops, etc.) , use of direct I/O to avoid unneeded data copying (i.e. use syscall file with O DIRECT flags),use of own prefetching/buffering mechanisms for overlapping I/O and computation (buf ostream and buf istream classes from the BM layer) and support of parallel disks.

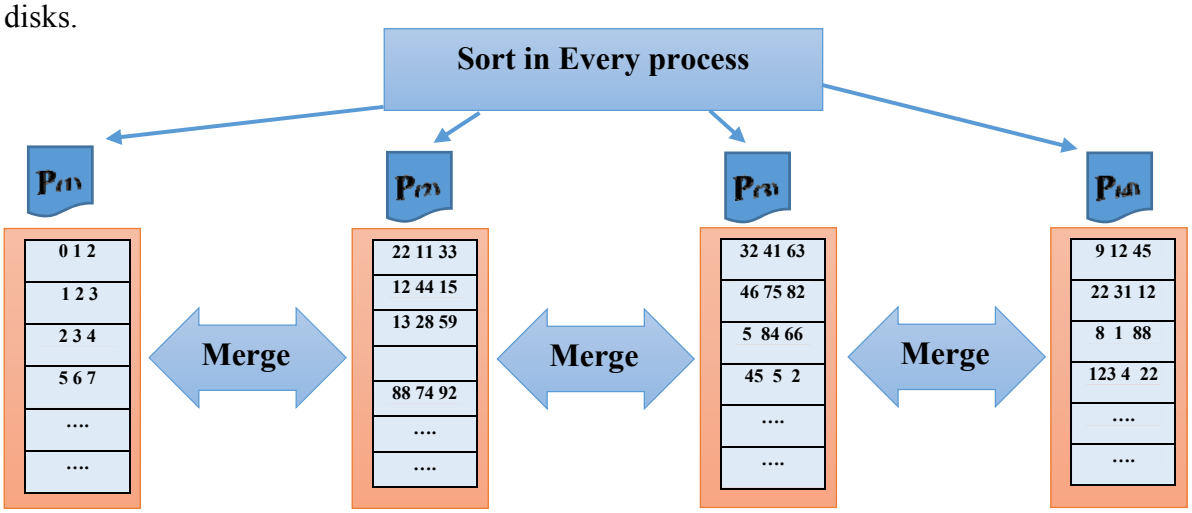

Figure 3.5 : Distributed sort STXXL Sort

3.2.4 Counting

Once the vectors are globally sorted, the counting becomes a trivial task. We scan through the vector and gather each identical consecutive entries into a single counted entry. Nevertheless, as the vectors are sorted globally and redistributed over the computing units, a problem may arise as the counting is carried on locally. The tail of the vector at given unit may correspond to the head of the vector at the next unit. We solve this issue by always passing the first entry in the vector, at any unit to the previous unit, except the first unit as it showed in Figure 3.5.

At the end of all this here we go to last step of our software is to count n-grams in our text .

In this and after we did a external sorting we begin to browse the vector till we have new element and we start to browse the next new element with calculated how much the element was repeated.

Which means that the sorting was just to make the vector easy to be read and count the n-grams in the text

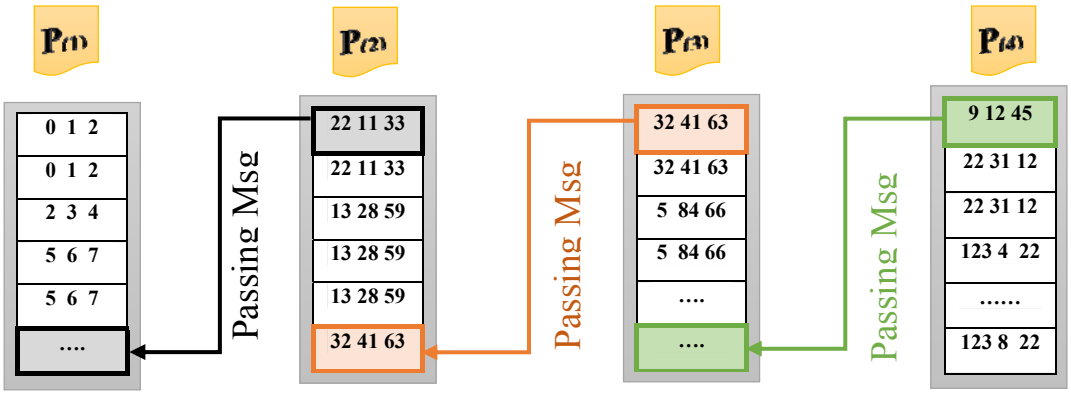

Figure 3.6: Solution to the problem which arises after distributed sorting: The same n-gram may exist on two different units

3.3 Parallel Computing

To make our work so fast we used more than 2 computer and for that we needed Parallel Computing implementation 'OPEN MPI' The Open MPI Project is an open source Message Passing Interface implementation that is developed and maintained by a consortium of academic, research, and industry partners.

The motivation that push us to use Advantage of MPI are: One of the oldest libraries, Wide-spread adoption. Portable and Minimal requirements on the underlying.

And in the Hardware side: Explicit parallelization, intellectually demanding, achieves high performance and Scales to large number of processors.

3.4 Shared memory parallelism

To have the ability for the processors to access all memory as global address space to share this memory, and provide more speed to our system we needed to use 'OPENMP', OpenMP is an API built for shared-memory parallelism. This is usually realized by multi-threading. The OpenMP API is comprised of three distinct components: compiler directives, runtime library routines, and environment variables.

Compiler directives: typed in your source code, these are instructions for the compiler regarding how to parallelize your code. If you set the right flags at compilation, these directives are read and understood, else, they are ignored, runtime library routines: typed in your source code, these are function calls to functions of the OpenMP library (omp.h in C/C++) ,environment

variable: typed in the terminal, this is a variable used by the system that can be modified or retrieved.

The motivation that push us to use OpenMp was : Easy to start, Data layout and decomposition are handled automatically by directives , Can incrementally parallelize a serial program – one loop at a time, Can verify correctness and speedup at each step, Provide capacity for both coarse-grain and fine-grain parallelism, and Significant parallelism could be achieved with a few directives

All those lead us to create a parallel computing network that will be configured and implemented as it showed in the next figure.

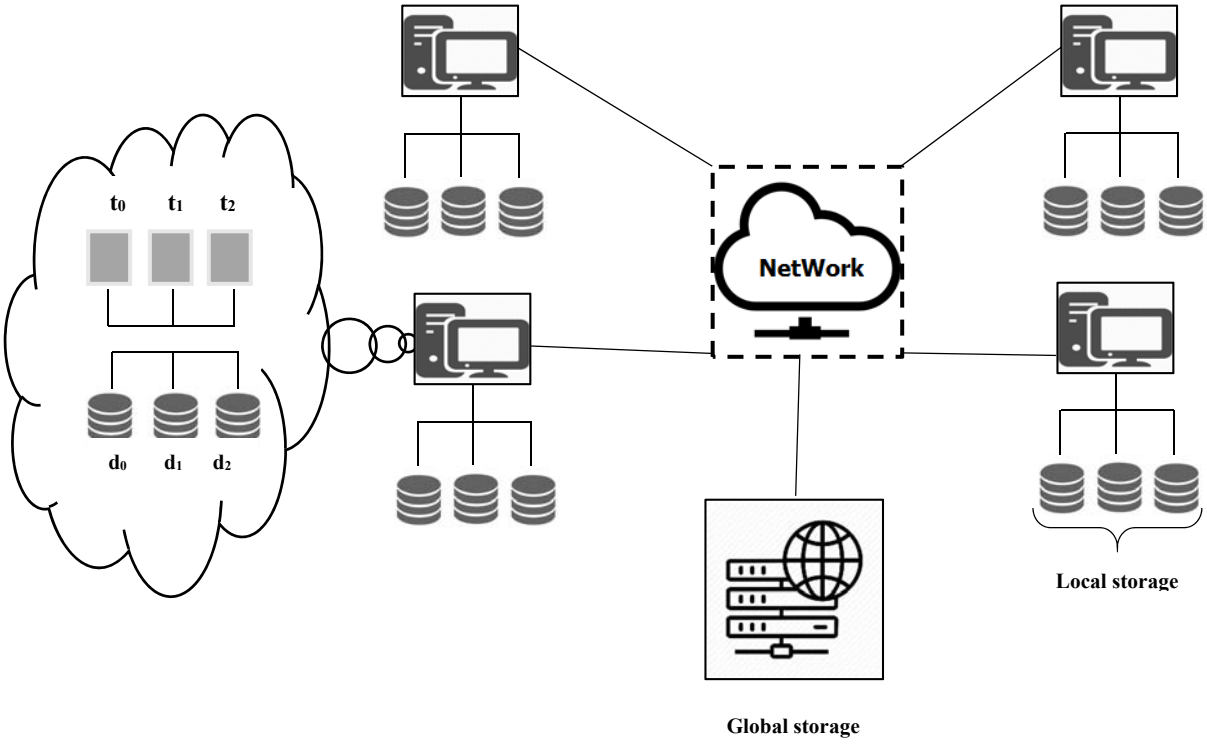

Figure 3.7: Typical hardware setup of our system

3.5 Architecture of the Proposed System:

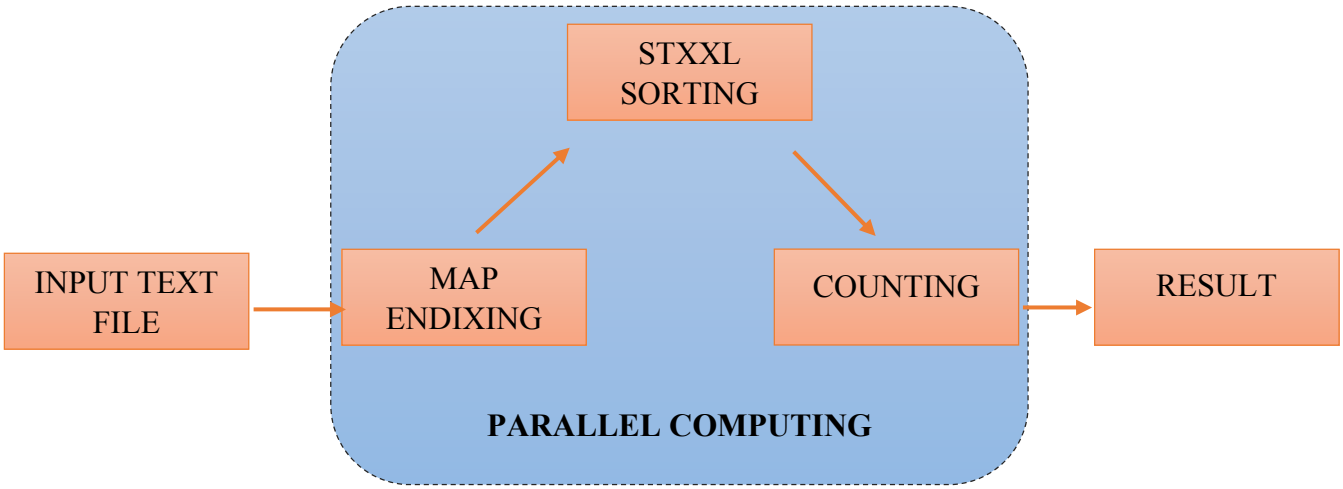

Figure 3.8 General scheme of our Counting system.

3.6 Working Environment:

3.6.1 Hardware Environment

In our experiments, we use tow laptop connected by a 10/100Mbs switch. And a couple of external disks .The characteristics of the PCs are shown in Table3.1 whereas the attributes of disks are in Table 3.2. The rates in the last table were computed using the Linux Gnome "Disks" utility.[1] The table gives the size of each disk, the corresponding average IO rates.

	Unit 0	Unit 1
Exploitation system	Linux mint18 cinnamon 64-bit	Linux mint18 cinnamon 64-bit
processor	Intel Celeron(R) CPU N3060 @ 1.60GHz	Intel(R) Core(TM) i5-3210M @ 2.50GHz
RAM	4 GB	6 GB
Hard disk	500GB	1000GB
Hard disk Rotation speed	5,400 rpm	5,400 rpm

Table 3.1: hardware properties

[1] https://github.com/GNOME/gnome-disk-utility

Disk	Disk Type	Disk Size 'GB' ↑	Average Read Rate 'Mb/S' ↑	Average Write Rate 'Mb/S' ↑
SD1	Internal hard Disk	500	95.8	-
SD2	Internal hard Disk	1000	91.1	-
SD3	External hard Disk	1000	27.3	22.0
SD4	USB Flash Disk	8	20.7	3.0
SD5	USB Flash Disk	8	24.5	6.1
SD6	USB Flash Disk	4	16.9	10.9
SD7	USB Flash Disk	4	18.8	3.2
SD8	USB Flash Disk	2	15.3	2.9
SD9	USB Flash Disk	8	8.3	2.2
SD10	USB Flash Disk	16	26.1	2.9

Table 3.2: Disk properties

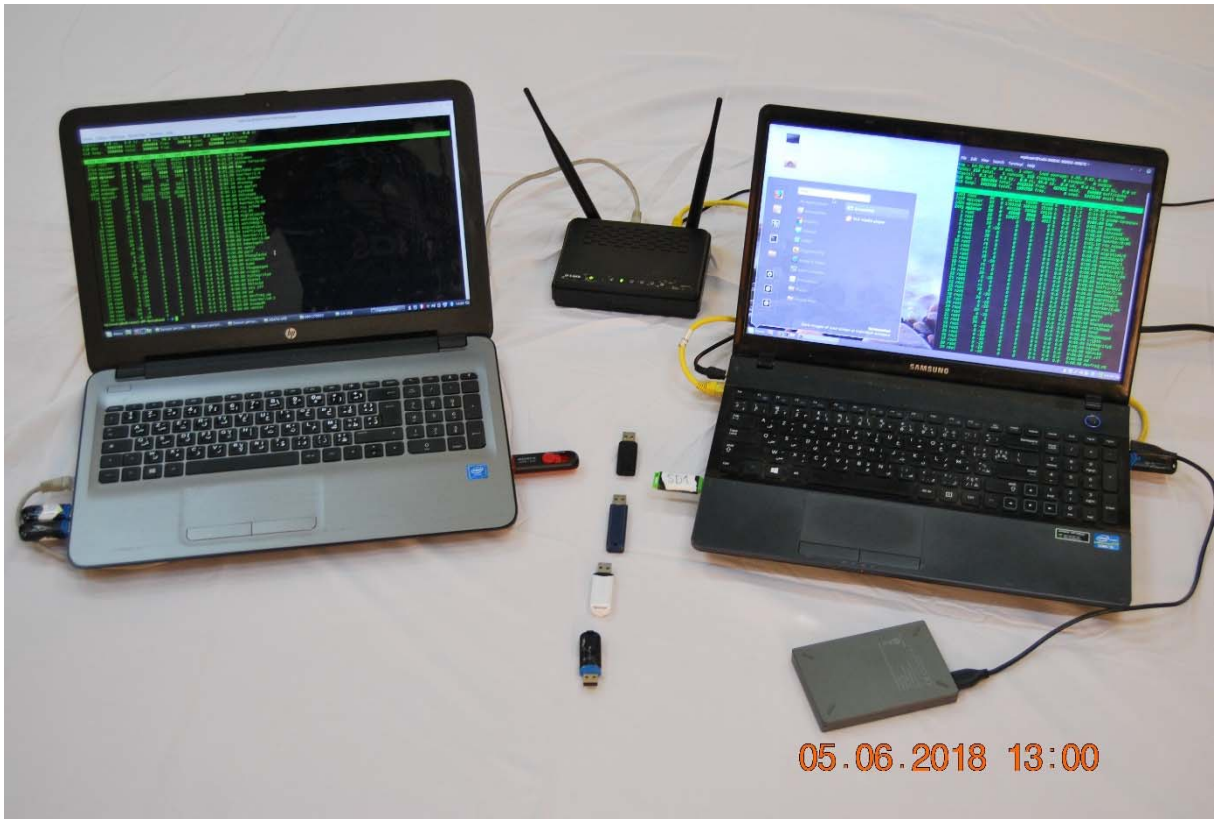

Figure 3.9: work environment

3.6.2 Corpora:

The work of our application is based in large corpora in this fact we chose to use Tatoeba corpora, the Content of Tatoeba in November 2017, the Tatoeba Corpus has over 6,000,000 sentences in 319 languages. The top 21 languages make up 90% of the corpus, [38]. Eighty-five of these languages have over 1,000 sentences. The top 13 languages have over 100,000 sentences each. The interface is available in 25 different languages. Parallel text corpora such as Tatoeba are used for a variety of natural language processing tasks which made us to choose this one. The Tatoeba data has been used as data for tree banking Japanese [36] and statistical machine translation [37].

In our experiments, we use a random sample of this corpus consisting of 414.5 thousand lines and 3.6 million words.

3.7 Results

Our software supports parallelism in many ways: on the CPU level, on the machine level, and on the disk level. We created different configurations to examine the efficiency of these parallelism levels. Table 3.3 records the results of all the experiments. The first column (Experiments) is an ID for the experiment. The second column shows the number of parallel threads per node (between brackets). The column “Hard Disks” shows the used disk IDs per node. Characteristics of these disks can b e retrieved from Table 3.2. In columns 4 through 6 the times of specific operations are given in seconds. The “Loading Time” (column 4) is the time spent by the system reading the corpus line by line, extracting n-grams from each line, indexing the underlying words (as described in Section 3.2.1), and pushing the indexed n-grams into the STXXL vector (as described in Section 3.2.2).

The “Sorting time” represents the time spent by all computing units in the distributed sorting. The “Total time” is the time of the whole process.

Experiments	Processor (Number of thread)	Hard Disks	Loading time 's' ↓	Sorting time 's' ↓	Total time 's' ↓
EXP1	Unit 0 (2)	SD1	6193.24	5.053	6229.96
	Unit 1 (2)	SD2	2455.46	5.618	6256.64
EXP2	Unit 0 (2)	SD1,SD3	6371.62	11.703	6441.32
	Unit 1 (2)	SD2,SD4	3033.99	53.065	6458.8
EXP3	Unit 0 (2)	SD1,SD3,SD6	6432.44	13.4365	6586.07
	Unit 1 (2)	SD2,SD4,SD7	2706.3	18.2532	6486.76
EXP4	Unit 0 (2)	SD5	7226.67	37.80	7326.37
	Unit 1 (2)	SD4	2596.01	26.63	7292.76
EXP5	Unit 0 (2)	SD5,SD6	6769.77	15.25	6847.29
	Unit 1 (2)	SD4,SD10	2609.77	13.80	6819.95
EXP6	Unit 0 (2)	SD5,SD6,SD9	6664.01	17.16	6715
	Unit 1 (2)	SD4,SD10,SD7	2510.87	15.23	6749
EXP 7	Unit 0 (2)	SD1	5882.96	5.92	6001
EXP8	Unit 0 (1)	SD1	5985.91	8.44	6107.06

Table 3.3: Table of our experimental result

We started with a preliminary experiment (EXP1) using the internal disks only of each machine. At first we thought that adding more disks will always result in a reduced total time. This was proved incorrect in the second experiment (EXP2). In EXP2, the process was much slower, even though we have more disks available to us. However, looking more closely to the disk configurations, we find out that this slowdown is mainly due to the fact that the added disks are much slower than the internal ones. This result is even confirmed further in the next row (EXP3): more slow disks result in even slower running time. The series of experiments (EXP 4 through EXP 6) demonstrate the effect of multiple parallel disks. With some minor exceptions, the execution time always diminish as the number of parallel external, disks. This behaviour is to be expected as STXXL assumes that the data is stripped over the different parallel disks. The following lines of experiments (EXP7 and EXP8) show the effect of shared memory parallelism. We kept the same disk configuration and changed the number of threads. This

experiment shows that the shared memory parallelism indeed helps, but with a much lower factor than the disk parallelism. The speedup in the first case is around 2 to 6% and in the second case it is only 1.7%. However, we have to mention that unfortunately the very heterogeneous setup we are using will have a negative impact on the results we draw.

3.8 Conclusions

In this chapter, we presented the details of the software developed in response to our original objective: perform counting on large corpora with limited hardware resources. By "large corpora" we mean both the total amount and the vocabulary size. The procedure is carried out as follows: all the data is indexed and then loaded into an STXXL vector. After that, this STXXL vector is sorted and the counts are generated.

Along this process, we support parallelism on multiple dimensions. First, the STXXL vectors are distributed over many computing units. Each of these computing units can perform its job using multiple parallel threads. Moreover, the STXXL IO latency can be improved by using multiple parallel disks on each of the computing units.

The features of our software were demonstrated through a series of experiments on real world corpora. The efficiency was quantified by reporting the amount of time the operation takes to complete in different hardware configurations. The experimentations show that although the shared memory parallelism has a positive impact of the performance, the gains obtained from parallelizing the IO are far more powerful.

CHAPTER 4
CONCLUSIONS

4.1 Conclusions and Future Work

Statistical natural language processing (NLP) is mainly based on models built from language data. These models will play a key role in the success of the corresponding NLP application. Exploiting the large amounts of language data available over the Internet, nowadays, is, on the one hand, necessary to produce robust and more accurate models. On the other hand, its availability in large amounts can present a real challenge to a resource constrained environment, as it might no longer fit into main memory during the training process.

Generally speaking, statistical modeling starts from the fundamental operation of counting the occurrences or cooccurrences of language units,such as n-grams or bilingual word pairs. The common technique of using dictionary (or map) data structures to perform this simple operation becomes impractical in large quantity scenarios. This last situation is overcome using external memory coupled with efficient sorting over the data items. However, the software products available on the market today and which use this approach to handle large data sizes do not exploit the parallel capabilities of the modern machines. We tried to answer this requirement through out the work presented in this thesis.

We have implemented a software package which performs n-gram counting over arbitrarily large corpora. The key new features in our solution is the fact that it, in addition to using external memories, exploits parallelism at the IO level as well as at the CPU level. The IO level parallelism is supported by building on top of STXXL data structures. STXXL library, unlike other common external memory solutions, implements a parallel disk model allowing parallel IO reads and writes. The parallelism is pushed even further by distributing the processing, which includes sorting and counting, over multiple machines. For that, we used an additional library, DEM_sort, written specifically for the STXXL vector data types. In addition to these libraries, we also used OpenMP and OpenMPI libraries. Open MPI handles the distributed parallelism and OpenMPsupports the shared memory parallelism.

Our experiments show that we almost always gain by using more disks, more threads, or more machines. However, it is difficult to conclude about a precise correlation between these parameters and the gained speedup. This difficulty is mainly due to the aggressive heterogeńeity

in the components used in our experimentation environment. The experiments were run on our personal laptops and a couple of usb flash disks. nor our laptops nor the usb disks have the same underlying properties. For instance, one of the laptops has more processing power and always completes much faster than the other. In such a heterogenous scenario, it would be useful to implement a load balancing algorithm, but this was out of our scope in this project. Another inconsistency can be seen by looking at the disks used in a given experiment. The introduction of a single slow disk will slow down the whole experiment.

As consequence, the first thing we are looking forward to is to perfectly homogenize our platform. We need to use a couple of identical machines together with similar external disks. We think this will give us more realistic and accurate conclusions about the system speed up. Another limitation we plan to overcome is the fixed n-gram order built into the source code and can be only changed by editing and recompiling the code. The experiments were carried out to compute 3-gram counts. For a 3-gram language model to be computed, we also need unigram and bigram counts. Once we solve this issue, we would continue by computing probabilities and outputting a complete language model.

REFERENCES

1- Baker, P. (2009) 'The BE06 Corpus of British English and recent language change'', International Journal of Corpus Linguistics 14(3): 312-37.

2- Marcel .D 2003 Encyclopedia of Library and Information Science (2nd edition) Edited by Miriam A. Drake

3- Noam Chomsky, 'Aspects of the Theory of Syntax', M.I.T. Press, 1969

4- H.K. Anasuya, Ganesh N,' Natural Language Processing: Going Ahead', IISc Bangalore, Karunya University Publication, 2010.

5- Davies, M.: The Corpus of Contemporary American English as the first reliable monitor corpus of English. Literary and linguistic computing (2010)

6- Allwood, J. 2007. Multimodal Corpora. In: Lüdeling, A. & M. Kytö (eds) Corpus Linguistics. An International Handbook. Mouton de Gruyter. Berlin: 207-225

7- Davies, M. 2008. The Corpus of Contemporary American English: 450 million words, 1990-present. URL: http://corpus.byu.edu/coca/

8- Louw, B. 1993. Irony in the Text or Insincerity in the Writer? The Diagnostic Potential of Semantic Prosodies. In Baker, M., Francis, G. & Tognini-Bonelli, E. (eds) "Text and Technology". Philadelphia/Amsterdam: John Benjamins.

9- McEnery, T., R. Xiao & Y. Tono. 2006. Corpus-Based Language Studies: An Advanced Resource Book.Taylor & Francis US.

10- Schiel F, C. Heinrich & S. Barfüßer. 2011. Alcohol Language Corpus. In: Language Resources andEvaluation, Springer, Berlin, New York, Vol 45.

11- James, G., Davison, R., Cheung, A., and Deerwater, S. 1994. English in computer science: a corpus-based lexical analysis. Hong Kong: Hong Kong University of Science and Technology and Longman Asia

12- Johansson, S., Atwell, E., Garside, R., and Leech, G. 1986. The tagged LOB corpus: Users' manual. Norwegian Computing Centre for the Humanities. Http://khnt.hit.uib.no/icame/manuals/lobman/INDEX.HTM.

13- Johansson, S. 1995. The approach of the Text Encoding Initiative to the encoding of spoken discourse. In Spoken English on Computer, eds. G. Leech, G. Myers and J. Thomas, 82-98. Harlow: Longman

14- Hofland, K., and Johansson, S. 1982. Word frequencies in British and American English. London: Longman

15- Hofland, K. c. 1999. ICAME CD-ROM. HIT Centre, University of Bergen. http://www.hit.uib.no/icame/cd.

16- Ide, N. 1996. Corpus encoding standard. Version 1.5. Expert Advisory Group on Language Engineering Standards (EAGLES). http://www.cs.vassar.edu/CES/.

17- Halliday, M. 1993. Quantitative studies and probabilities in grammar. In Data, description discourse, ed. Michael Hoey, 1-25. London: Harper Collins

18- Halteren, H. v. ed. 1999. Syntactic wordclass tagging. Text, speech, and language technology; 9. Dordrecht; Boston: Kluwer Academic Publishers

19- Hirst, D. 1991. Intonation models: towards a third generation. In Actes du XIIeme Congres International des Sciences phonetiques. 19-24 aout 1991. Aix-en-Provence, France, 305- 310. Aix-en-Povence: Universite de Provence, Service des Publications

20- Marcus, M., Santorini, B., and Marcinkiewicz, M. 1993. Building a large annotated corpus of English: the Penn Treebank. Computational Linguistics 19:313-330.

21- Roach, P., and Arnfield, S. 1995. Linking prosodic transcription to the time dimension. In Spoken English on Computer, eds. G. Leech, G. Myers and J. Thomas, 149-160. Harlow: Longman.

22- Kenneth Heafield. KenLM: faster and smaller language model queries. In Proceedings of the EMNLP 2011 Sixth Workshop on Statistical Machine Translation, pages 187–197, Edinburgh, Scotland, United Kingdom, July 2011. URL http://kheafield.com/professional/avenue/kenlm.pdf.

23- Demetris.Z EPL446 – Advanced Database Systems Chapter 13: Ramakrishnan & Gehrke pp.1

24- Donald Knuth, The Art of Computer Programming, Volume 3: Sorting and Searching, Second Edition. Addison-Wesley, 1998, ISBN 0-201-89685-0, Section 5.4: External Sorting, pp.248–379.

25- Ellis Horowitz and Sartaj Sahni, Fundamentals of Data Structures, H. Freeman & Co., ISBN 0-7167-8042-9.
26- Vaccari C. (2014). Big Data and Official Statistics. PhD Thesis, School of Science and Technologies - University of Camerino, available at https://www.academia.edu/7571682/PhD_Thesis_on_Big_Data_in_Official_Statistic.
27- John Wiley & Sons, Ltd. «STXXL: standard template library for XXL data sets.» SOFTWARE—PRACTICE AND EXPERIENCE, 2007.
28- Roman Dementiev, Lutz Kettner, Peter Sanders. STXXL. 2007. http://stxxl.org/tags/1.4.0/introduction.html (accès le 06 26, 2018).
29- R. Dementiev and P. Sanders. Asynchronous parallel disk sorting. In 15th ACM Symposium on Parallelism in Algorithms and Architectures, pages 138–148, San Diego, 2003.
30- R. D. Barve, E. F. Grove, and J. S. Vitter. Simple randomized mergesort on parallel disks. Parallel Computing, 23(4):601– 631, 1997.
31- S. Rajasekaran. A framework for simple sorting algorithms on parallel disk systems. In 10th ACM Symposium on Parallel Algorithms and Architectures, pages 88–98, 1998.
32- Rezaul Alam Chowdhury and Vijaya Ramachandran. The cache-oblivious gaussian elimination paradigm: theoretical framework and experimental evaluation. In SPAA '06: Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures, pages 236–236, New York, NY, USA, 2006. ACM Press.
33- Amit Mhatre and Piyush Kumar. Projective clustering and its application to surface reconstruction: extended abstract. In SCG '06: Proceedings of the twenty-second annual symposium on Computational geometry, pages 477–478, New York, NY, USA, 2006. ACM Press.
34- Deepak Ajwani. Design, Implementation and Experimental Study of External Memory BFS Algorithms. Master's thesis, Max-Planck-Institut fu¨r Informatik, Saarbru¨cken, Germany, January 2005.
35- A. Crauser and K. Mehlhorn. LEDA-SM, extending LEDA to secondary memory. In 3rd International Workshop on Algorithmic Engineering (WAE), volume 1668 of LNCS, pages 228–242, 1999.
36- 117. Francis Bond, 栗林 孝行 [Takayuki Kuribayashi], 橋本 力 [Hashimoto Chikara] (2008) HPSGに基づくフリーな日本語ツリー バンクの構築 [A free Japanese Treebank based on HPSG]. In 14th Annual Meeting of The Association for Natural Language Processing, Tokyo.
37- Jump up^ Eric Nichols, Francis Bond, Darren Scott Appling and Yuji Matsumoto (2010) Paraphrasing Training Data for Statistical Machine Translation. Journal of Natural Language Processing, 17(3), pages 101–122.
38- "Tanaka Corpus". EDRDG Wiki. Electronic Dictionary Research and Development Group. 3 February 2011. Retrieved 20 March 2011.
39- Jump up^ Breen, Jim (2 March 2011). "WWWJDIC – Information". WWWJDIC. Monash University. Retrieved 20 March 2011.