Experimental Analysis of a Parallel Quicksort-Based Algorithm for Suffix Array Generation

Autran Macêdo[1], Marco Antônio Cristo[1], Elaine Spinola Silva[1], Denilson Moura Barbosa[1], João Paulo Kitajima[1], Berthier Ribeiro[1], Gonzalo Navarro[2], and Nivio Ziviani[1]

[1] Departamento de Ciência da Computação
Universidade Federal de Minas Gerais
Belo Horizonte, MG - BRAZIL
latin@dcc.ufmg.br
[2] Departamento de Ciencias de la Computación
Universidad de Chile
Santiago - CHILE
gnavarro@dcc.uchile.cl

Abstract. This paper presents experiments performed with an implementation of a quicksort-based parallel indexing algorithm. Besides the expected reduction in execution time, it was observed that the word frequency distribution of the input textual database has a strong influence on performance. Communication and computational load balances are achieved by processing the same quantity of text on each processor. This effectively occurs due to the auto-similar feature of texts, verified experimentally in this work. Also, as seen by the experiments, the auto-similarity of the word frequency distribution implies that this distribution is independent of the text size. In terms of implementation, the knowledge *a priori* of this word frequency may improve the indexing time by eliminating certain parts of the algorithm.

Keywords: Parallel Processing, Information Retrieval, Index Generation, Auto-Similarity, Message Passing.

1 Introduction

Information retrieval is a research area of growing interest by the scientific community. One of the most relevant research field in that area is the string search in textual databases. This string search involves not only the database query, but also the database indexing and the user interface [1]. In the case of information retrieval in Internet homepages, the search process may involve also the automatic scan of World Wide Web (WWW) sites and the download of these homepages for further indexing.

Index generation time is critical. It has at least the sequential complexity in the order of the database size, since all the words may be indexed. In this

sense, parallel strategies can be devised in order to reduce the index generation time. The algorithm proposed in [2] is based on a quicksort approach, where the textual database is partitioned through processors interconnected by a fast network [3]. The index structure is based on a suffix array [1].

The goal of this paper is to present some experimental results of the first version of the algorithm implementation. It was observed that the word frequency distribution in textual databases plays an important role on the program performance. The following Section describes the parallel index generation algorithm. Next, experiments on index generation are presented. The influence of text characteristics is discussed and followed by some conclusions.

2 The Parallel Generation of Suffix Arrays

Searching a large full text for user specified patterns is a time consuming task which requires special indexing schemas. A *suffix array* (or PAT *array*) [1] is a linear structure composed of pointers to every suffix in the text (since the user normally is allowed to query on words, it is customary to index only word beginnings). These index pointers are sorted according to a *lexicographical ordering* of their respective suffixes and each index pointer can be viewed simply as the offset (counted from the beginning of the text) of its corresponding suffix in the text. To find the user patterns, binary search is performed on the array.

The central idea of the parallel algorithm is as follows, considering a fast network of independent computers [3]. Imagine the final result of the process: the global sorted suffix array. If that array is cut in b similarly-sized portions (which is called *slices*), what the algorithm does is to assign a slice to each processor and make it sort that slice. Originally, each processor contains some elements of each slice.

An α-percentile is the value at position αn in the global sorted suffix array. For example, the $1/r$-percentile is the element at position b. Our algorithm partitions the data to be worked on by each processor by finding the percentiles $1/r$, $2/r$, ..., $(r-1)/r$ (r is the number of processors). An alternative definition for slice is: the portion of the global suffix array between two consecutive (i/r)-percentiles.

The algorithm proceeds in four steps:

- **Step 1** - One master processor splits the text into pieces of same size and distributes them among slave processors;
- **Step 2** - Each processor builds internally its local suffix array and determines its local percentiles;
- **Step 3** - The processors cooperate to find the r global percentiles. This defines the part of each slice stored at each processor;
- **Step 4** - The processors engage in a distribution process so that every processor gets the part of its slice stored on any other processor;
- **Step 5** - Each processor completes internally the sorting of its slice.

Consider a text T. T is split into pieces of the same size according with the number of processors involved in the generation of the index, so that each processor has a part of the text (step 1). Each processor generates its local suffix array (step 2). In order to generate the global suffix array, the processors find the percentiles of its local suffix array and broadcast them to the others to determine the global percentiles (step 3). After that, each processor is able to know which part of its suffix array belongs to itself and which parts belongs to their partners. The global all-to-all communication is performed (step 4). Finally, each processor sorts its local suffixes (step 5). The concatenation of local suffix array (of all processors) leads to the global index suffix array.

It can be noticed that the steps 4 and 5 are time dominant, due to suffix arrays broadcast and the need of I/O operations. Figures 1 and 2 show this fact. Figure 1 presents two turning points (phases 1 and 5 at 8 processors) due to contention of disk. It can be argued why not just parallelize only these steps. The answer is the availability of primary memory. The aim of this study is the index generation of very large files (order of GigaBytes). A single computer to perform steps 1, 2 and 3 could not support all text in primary memory, without avoiding page faults.

In this way, the complexity of the algorithm (see [2] for details), in the average case, is

$$O(b \log n)\mathbf{I} + O(r^2 \log b + b)\mathbf{C} \quad = \quad O(b \log n)\mathbf{I} + O(b)\mathbf{C}$$

where n is the text size and b is the slice size. $\mathbf{I}$ is the computation unit cost and $\mathbf{C}$ is the communication cost.

3 Experimental Analysis

The following Sections present the experimental environment and measures concerning execution time and load balancing. Parallel quicksort is typically a scalable strategy: a reduction of the execution time is expected. However, for parallel indexing, the characteristics of the input textual database also influence strongly the program performance.

3.1 Experiments

Two message passing parallel machines are being used in the implementation of the algorithm. One machine is in the CENAPAD - MG/CO, a brasilian supercomputer center, located at UFMG (Federal University at Minas Gerais); the other one is in the LMC (Laboratory of Modeling and Calculus), located at IMAG (Mathematics Institute of Grenoble) in France. The computer of CENAPAD - MG/CO is an IBM SP with 41 nodes and 48 processors at 120 MHz with memory ranging from 256 MegaBytes (MB) to 1 GigaBytes (GB) of primary memory. The network is a switch at 155 MB/s and all processors share a single disk system. The computer in LMC is an IBM SP with 32 processors at

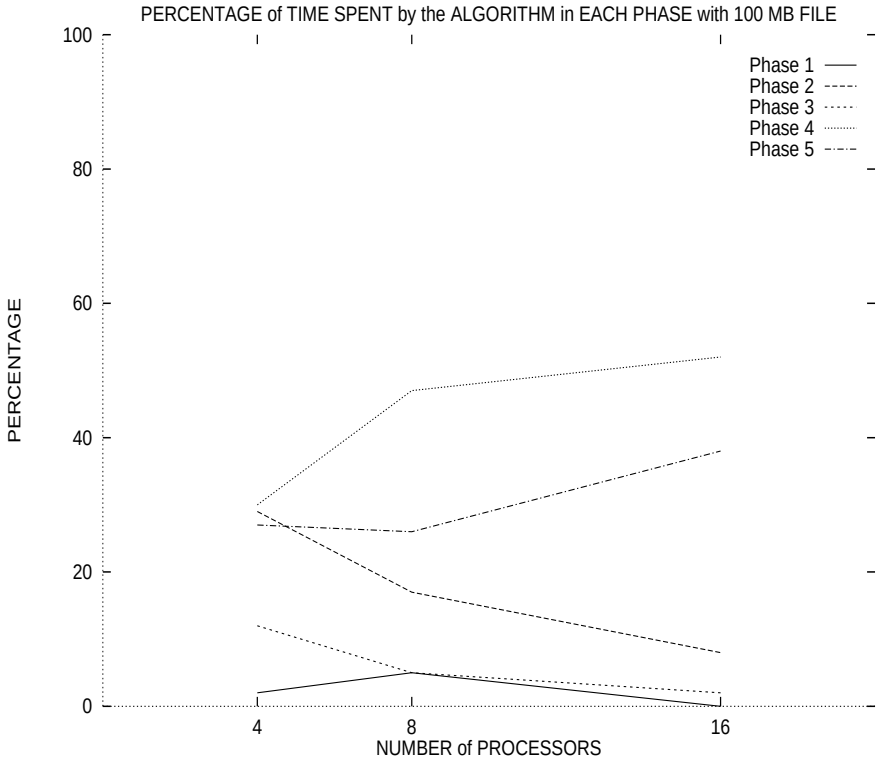

Fig. 1. Percentage of time spent by each step of the program, when it was submmited to a network of workstations in CENAPAD - MG/CO with a text input file of 100 MB.

66 MHz and 64 MB of main memory. The network has a switch at 40 MB of unidirectional bandwidth and a local disk for each processor.

The parallel program is written in ANSI C using MPI (Message Passing Interface) [4] as communication library. The benchmark textual database is composed of file texts of 100 MB and 200 MB, extracted from the Wall Street Journal of TREC-3 collection [5].

The experiment results presented in this article were obtained by executing the program in CENAPAD - MG/CO considering 1, 4, 8, and 16 processors. Some details should be stated:

- only 54 MB of main memory were used by the processors of CENAPAD - MG/CO. This limitation was set because it is the maximum portion of main memory used by the LMC processors, when experiments are performed in France. This memory size compatibility have to be kept because memory is important in this study;
- experiments with 2 processors were not performed due to the politic of memory utilization adopted by the program. By this politic, $\frac{2}{3}$ of main memory

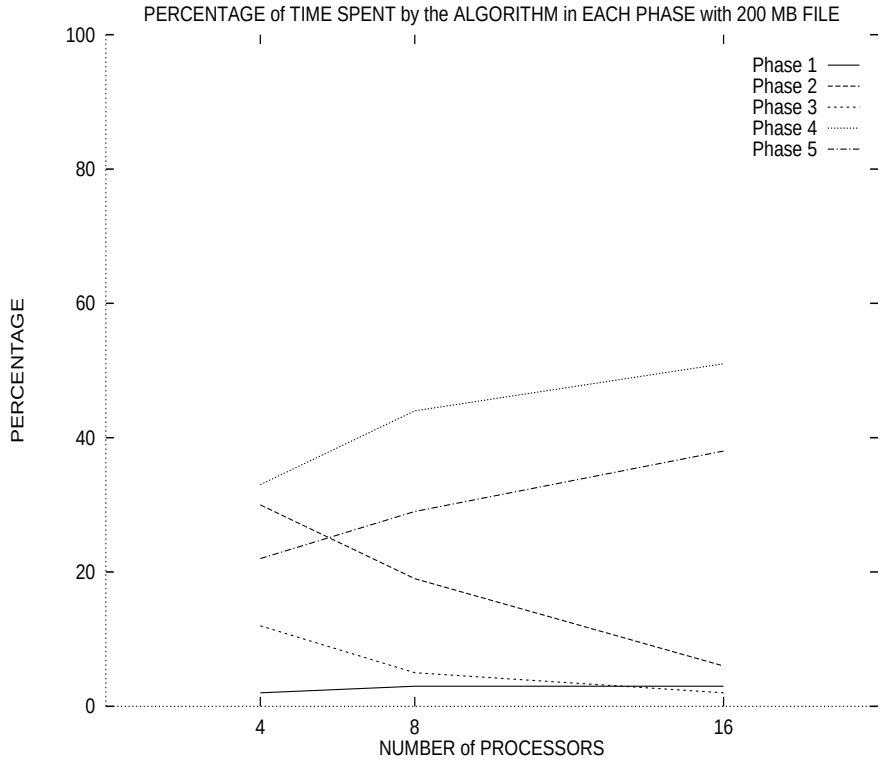

Fig. 2. Percentage of time spent by each step of the program, when it was submmited to a network of workstations in CENAPAD - MG/CO with a text input file of 200 MB.

is left to the PAT array and the other $\frac{1}{3}$ is left to the text. Two processor would have too many I/O disk operations, what would be a similar case of the sequential algorithm implementation.

Figures 3 and 4 present the performance of the program, considering text files of 100 MB and 200 MB. The speedup was measured considering an sequential implementation of the algorithm presented in [6].

It can be noticed (figure 4) that with 4 processors speedup is super-linear (5.12 and 5.76 with files of 100 MB and 200 MB, respectively) and with 16 processors speedup is bad. Super-linear speedup is observed because the implementation of the sequential algorithm has a quadratic behaviour due to I/O disk operations, although its complexity is $n \log n$ [6]. Sub-linear speedup occurs because of the single disk shared by every processors in CENAPAD - MG/CO. The more processors involved in the index generation, the greater is the number of files that each processor must deal with. These files are created by the processors during the broadcast (step 4). The competition by I/O bus and seek time of the disk determine the low performance of the program, in this envi-

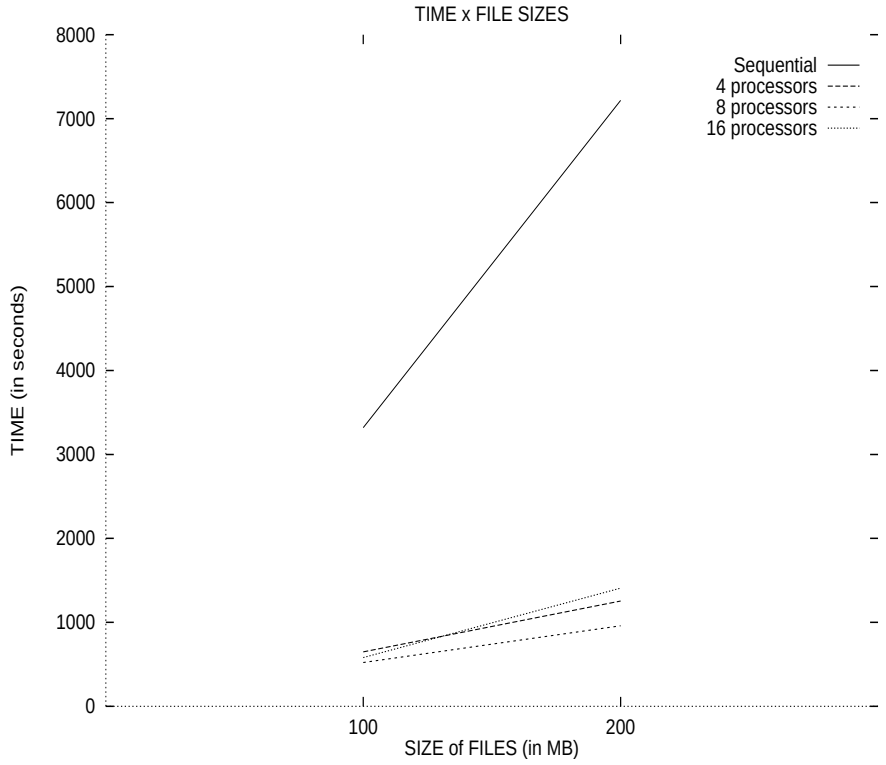

Fig. 3. Measures of elapsed time for index generation using 1, 4, 8 and 16 processors, connected by fast network at CENAPAD - MG/CO, considering files of different sizes.

ronment. The domination of I/O time in the performance of program can be observed in figures 5 and 6. The curves show the percentage of time spent in step 4 of the algorithm. It can be devised that the percentage of time in I/O operations is increasing, as increases the number of processors. There are 3 curves: communication, I/O operations, and others. This last curve reflects activities like:

- suffix array compression;
- package of data to be transmitted;
- contention in the suffix array transmission.

Figures 5 and 6 present also a turning point on curve "others", when the number of processors is 8. This turning point is caused by the competition of the processors for the disk. Besides, the graphics show the percentage of time spent by the algorithm when communication, I/O disk operations, and other factors are considered. Beyond 8 processors, the percentage of time of other factors (curve "others") decrease, on the hand the percentage of time of I/O disk operations increase.

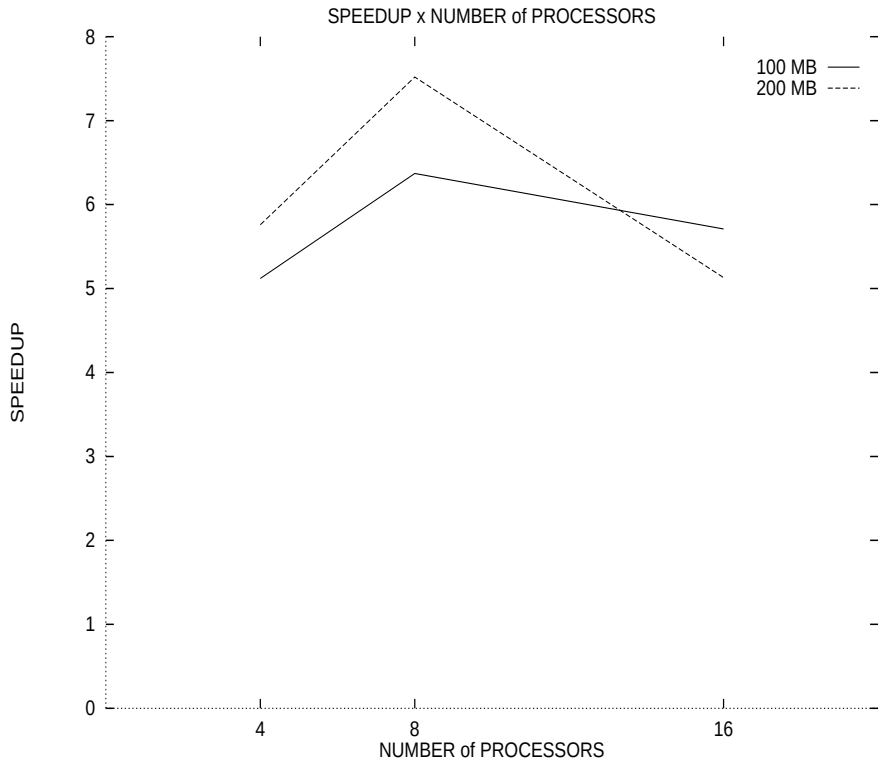

Fig. 4. Curves of speedup obtained in a network of workstations at CENAPAD - MG/CO.

3.2 Influence of Text Characteristics

One concern with the algorithm here presented is the load balance during steps 4 and 5. More specifically, in the step 4, a non homogeneous communication load occurs if exchanged *parts* of slices are of different sizes. In step 5 also, the resulting slice to be sorted locally would have different sizes for different processors. This would imply a non balanced computational load. This happens when the word frequency distribution is not *auto-similar*.

A structure is said *strictly auto-similar* [7] if it can be recursively decomposed in small pieces where each one is a replica of the original structure. It is important to say that these parts are obtained through a *scale transformation* of the original structure. Those structures that can be decomposed in similar parts *until a given scale* is said *auto-similar*.

In order to detect auto-similarity, experiments were done over the following collections [5] *AP* (Associated Press) (1988), *WSJ* (1987), and *Ziff-Davis* (complete). The collection files were split into pieces of $1,000$; $10,000$; $100,000$;

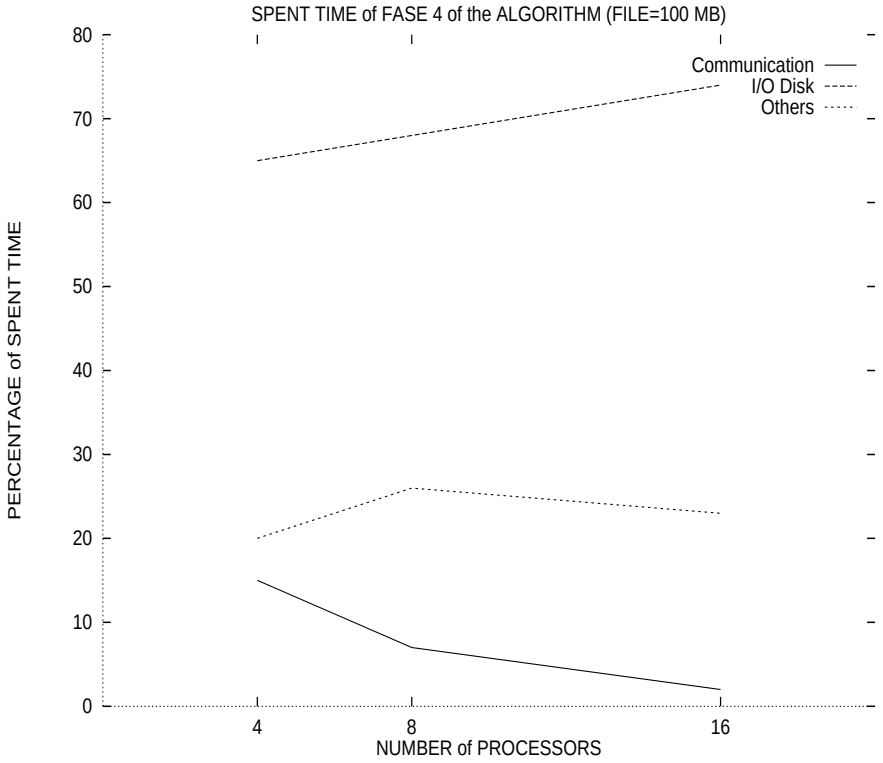

Fig. 5. Percentage of time spent in step 4 of the algorithm (which includes communication, I/O operations on disk, compression of suffix arrays, packaging of data to be transmitted, contention in suffix array transmission), considering a file of 100 MB.

300,000; 550,000; 700,000; 850,000; and 1,000,000 words. These diffrente sizes were to detect if the number of words in files of different sizes grows linearly.

Figures 7 and 8 describe the number of words starting with a given letter *versus* the alphabet letters, independently of the text file size. Since these graphics have the same shape, they are similar, from the geometric point of view. This similarity occured in all files collections experienced.

Table 1 presents linear regression information concerning each alphabet letter for the AP collection. The numbers in this table confirm that the number of words starting by a given letter ℓ grows linearly with the text file size. The coefficient of determination of the linear regression is close to 1 (see third column). Similar results were obtained in Wall-Street Journal and Ziff-Davis Collection.

The second column of Table 1 (times 100) can be also considered as the percentage of words starting with a given letter ℓ in a text file of size t. Ignoring round errors, the total sum of the numbers of the second column of this table is 1.

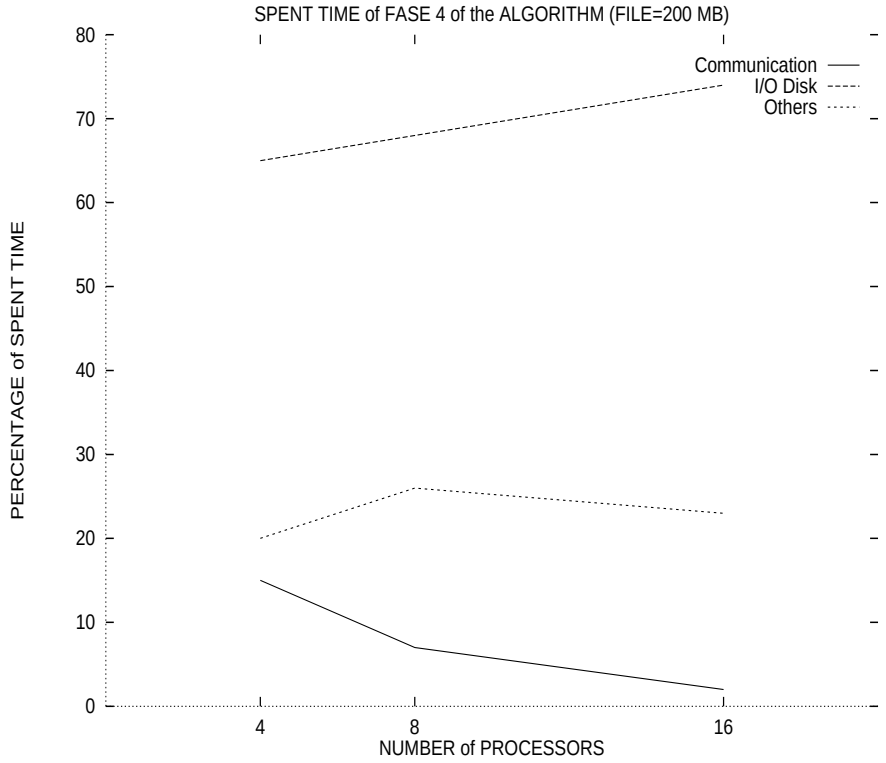

Fig. 6. Percentage of time spent in step 4 of the algorithm (which includes communication, I/O operations on disk, compression of suffix arrays, packaging of data to be transmitted, contention in suffix array transmission), considering a file of 200 MB.

On a probabilistic point of view, Table 2 presents the distribution function of the word frequency. Due to the auto-similarity, this function is independent of the text file size. For example, the probability of choosing a word starting by letters A or B or ... or K is around 50%, independent of the file size.

Experimentally, the text auto-similar feature shows that the computed global percentiles effectively generates a homogeneous distribution of suffix pointers among the processors (step 4: communication load balance). The number of bytes sent and received by each processor is almost the same for all processors. Consequently, the final local sort (step 4) will work with roughly the same number of pointers, implying a computational load balance. The same conclusion was obtained by simulation of the algorithm [2].