Universidade Federal de Minas Gerais

Instituto de Ciências Exatas

Departamento de Ciência da Computação

Recursive Parallel Generation of Suffix Arrays

Gonzalo Navarro
Berthier Ribeiro-Neto
João Paulo Kitajima
Nivio Ziviani

Relatório Técnico 010/97[1]

April 1997

[1] Available from `http://www.dcc.ufmg.br/~nivio`

Recursive Parallel Generation of Suffix Arrays

Gonzalo Navarro
Departamento de Ciencias de Computación
Universidad de Chile
Chile

Berthier Ribeiro-Neto[†]
João Paulo Kitajima[‡]
Nivio Ziviani[§]
Departamento de Ciência da Computação
Universidade Federal de Minas Gerais
Brazil

Abstract

We present an algorithm for the distributed computation of suffix arrays for large texts. The parallelism model is that of a set of sequential tasks which execute in parallel and exchange messages among them. The subjacent architecture is that of a high-bandwidth network of workstations. In such a network, a remote memory access has a transfer time similar to the transfer time of magnetic disks (with no seek cost) which allows using the aggregate memory distributed over the various workstations as a giant cache for disks. Our algorithm takes advantage of this architectural feature to implement a quicksort-based distributed sorting procedure for building the suffix array. We demonstrate that such algorithm has time complexity given by $O(n \log^2 r \; / \; r)$ where n is the text size and r is the number of processors. This is considerably faster than the best known sequential algorithm for building suffix arrays which has time complexity given by $O(n^2 \; / \; m)$ where m is the size of the main memory. Further, we demonstrate that our algorithm scales up much nicer than a previously proposed distributed algorithm based on the mergesort sorting procedure.

1 Introduction

The advent of powerful workstations has allowed the consideration of alternative models for information retrieval other than the traditional one of a collection of documents indexed by keywords. One such a model which is gaining popularity is the *full text* model. In this model documents are represented by either their complete full text or extended abstracts. The user expresses his information need by providing words, phrases or patterns to be matched for and the information system retrieves those documents which contain the user specified strings.

[†]This author has been partially supported by CNPq Project 300188/95-1.
[‡]This author has been partially supported by CNPq Project 300815/94-8.
[§]This author has been partially supported by CNPq Project 520916/94-8 and Project RITOS/CYTED

While the cost of searching the full text is usually high, the model is powerful, requires no structure in the text nor guessing the potential queries at indexing time, and is conceptually simple [4].

To reduce the cost of searching a full text, specialized indexing structures are adopted. The most popular of these structures are the *inverted lists*. Inverted lists are useful because their searching strategy is based on the vocabulary (i.e., the set of distinct words in the text) which is usually much smaller than the text and thus, fits in main memory. However, the lists containing the positions in which each word appears in the text are large and as a result the whole structure takes space which is close to the text size. *Suffix arrays* [7] or *pat arrays* [3, 4] are more sophisticated indexing structures which also take space close to the text size. Their main drawback is its costly construction and maintenance procedure (i.e., creating and updating a suffix array takes time). However, such arrays are superior to the inverted lists for searching phrases or complex queries [7, 4]. For instance, a search for the composed string "*transaction concurrency control in relational databases*" is faster with a suffix array than with an inverted list.

The emergent fast switching technology has allowed the dissemination of high speed networks at relatively low cost. In such networks, one can view the workstations as processing nodes (with local memory) and the switching equipment as an interconnectivity mesh of a *virtual* distributed parallel machine. Such view is reasonable because most personal workstations are completely idle most of the time [1] and because the latency of these high speed networks is low. For efficient parallelism, however, low latency is not the only requirement. Besides low network latency, the overhead involved in packing, shipping, and unpacking messages must be small in a user-to-user message exchange. This overhead is CPU time which is unavailable for computation. To reduce this overhead one has to deal with the delays imposed by the operating system (i.e., message processing, layered architecture, etc). Such task involves the redesign of a portion of the operating system and the design of specialized network hardware. While such designs are not a reality yet, efforts are under way [1, 2].

Therefore it is reasonable to think of the workstations in a high speed network as the nodes of a distributed parallel machine. The driving argument for such consideration is that the parallel machine comes for free in the current quest for fast hardware and speedy communication. In such parallel machine, accessing the memory of remote nodes has a transfer delay which is very similar to the delay for reading data from a local disk at transfer rate [9, 6]. Thus, in this *virtual* distributed parallel machine we can think of the aggregate distributed memory as a giant cache for disks.

The purpose of this paper is to investigate a new algorithm for the distributed parallel generation of large suffix arrays in the context of a high-bandwidth network of workstations. The motivation is twofold. First, the high cost of the best known sequential algorithm (for suffix array generation) leads naturally to the exploration of parallel algorithms for solving the problem. Second, the usage of a network of workstations (connected by a fast switch) as a parallel machine is an attractive alternative nowadays [1]. The distributed algorithm we propose is based on a parallel quicksort [8]. The algorithm is an alternative to a previously proposed mergesort-based distributed algorithm [9, 6]. We show that the quicksort-based algorithm is faster and, more important, that it scales up well while the mergesort-based algorithm does not.

2 Basic Definitions

In full text retrieval, the entire text is viewed as one very long string. In this string, each position k is associated to a semi-infinite string which initiates at position k in the text and extends to the right as far as needed to make it unique. Such semi-infinite strings are called *sistrings* [4]. The user specifies his information need by referring to the sistrings he is interested in. The task of the information system is to search the full text for the occurrences of the sistrings which satisfy the user specified need. In practical situations, however, users are usually interested in finite sequences of characters and not in sistrings. A reasonable compromise is to let the user refer to *pruned sistrings* instead, where the prunning occurs at a length of ℓ characters. For practical matters, a value of 48 for the length ℓ of the pruned sistrings is more than reasonable [3]. Therefore, in the rest of the paper we consider that the user is interested in pruned sistrings whose length ℓ does not exceed 48 and that the task of the information system is to search the full text for the occurrences of these user specified pruned sistrings. For simplicity, throughout the paper we to refer to *pruned sistrings* as *sistrings*. It should always be clear that the sistrings are pruned at a length ℓ.

Searching a large full text for the user specified sistrings is a time consuming task which requires special indexing schemas. One of these indexing schemas is the suffix array. A *suffix array* (or *pat array*) is a linear structure composed of pointers (here called *index pointers*) to every sistring in the text. These index pointers are sorted according to a *lexicographical ordering* of their respective sistrings and each index pointer can be viewed simply as the offset (counted from the beginning of the text) of its corresponding sistring in the text. Further, we consider that the user is mainly interested in text words and/or text phrases such that the suffix array needs to include solely index pointers to those sistrings which occur at the beginning of words in the text.

From the above, we conclude that the building of a suffix array is simply an indirect sorting of the index pointers. The difficult part is to do this sorting efficiently when large texts are involved (i.e., texts of size 1G (giga) bytes or more). Large texts do not fit in main memory and an external sorting procedure has to be used. The best known sequential procedure for generating large suffix arrays takes time proportional to $O(n^2\ /\ m)$ where n is the text size and m is the size of the main memory [4]. In this paper, we explore the distributed parallel generation of large suffix arrays in the context of a high speed network where the attached workstations are viewed as processing nodes of a distributed parallel machine. Our idea is to use the aggregate distributed memory of this machine to hold the text. Accessing this aggregate memory requires frequent accesses to remote data (accross the network) which take time similar to the time to get data from a local disk at transfer rate [9, 6]. However, this distributed parallel machine gives us with two critical advantages. First, the access time does not improve but the aggregate memory allows random access which we do not have with the local disk. Second, we can split our problem in smaller parts and work on these in parallel.

The underlying high speed network could be, for instance, an ATM network running at 155 Mbps. In an ATM network, all workstations are connected to a central ATM switch (hierarchies are possible but are not considered here) which runs internally at a rate much higher than the external 155 Mbps. Any pair of nodes can communicate at a guarantteed rate of 155 Mbps without contention. Further, the communication between machines A and B does interfere with the communication between machines C and D and broadcasting can be done easily. The communication protocol is the ATM protocol which exchanges packets of size 48 bytes.

3 The Previous Mergesort-Based Distributed Algorithm

This algorithm is based on a distributed hierarchical mergesort of the index pointers [9, 6]. First, the whole text is broken up in smaller text blocks (each of size b) which fit the main memory of any workstation. Second, these text blocks of identical size are distributed over the high-bandwidth network to the participating workstations. Third, each workstation generates (in parallel) a suffix array for its block of text. Fourth, these partial suffix arrays are merged together to form the final suffix array. This merging phase is the more involving task and is discussed in more detail in the immediately following.

The merging phase is based on a distributed and hierarchical procedure. At the first level, adjacent processors are grouped in a pairwise basis. The processors in a pair exchange index pointers (and the respective sistrings) to merge their suffix arrays in a combined suffix array which spans the memories of both processors. At the second level, each pair of processors combines its joint suffix array with that of another pair of processors to form a resultant suffix array which spans the memory of four processors. This merging task is repeated until a single resultant suffix array is generated. This is the suffix array for the whole text.

In [9, 6] we show that this mergesort-based algorithm presents a time complexity given by $O(n)$, where n is the text size. This is considerably better than the sequential algorithm discussed in [4] for which the time complexity is $O(n^2/M)$ where M is the size of the available memory. Despite this improvement in time complexity (which translates to gains in real time performance of an order of magnitude or more [9, 6]), this mergesort-based distributed algorithm does *not* scale up. For instance, if we double the size n of the text and also double the number of processors available, we observe that the expected running time of the algorithm also doubles. This is unfortunate because we expect the running time to remain constant in such situation.

In the following section we introduce a quicksort-based alternative for the above algorithm which performs far better and which scales up nicely.

4 The Quicksort-Based Distributed Algorithm

Our quicksort-based algorithm also utilizes the aggregate memory as a giant cache for *disk*. However, instead of creating groupings which become larger and larger (as in the mergesort-based algorithm), the quicksort-based algorithm starts with one large group (which includes all processors) and splits this group in subgroupings which become smaller and smaller. While the mergesort-based algorithm works his way up the *grouping* tree (i.e., a tree which represents the groupings of processors) the quicksort-based algorithm works his way down the grouping tree. As a consequence, for the quicksort-based algorithm, doubling the size of the problem adds a single additional level at the bottom of the grouping tree. Since levels at the bottom of the tree are cheap to execute (because of the higher degree of parallelism), the running time increases only slightly. Thus, the running time of the algorithm does not double as it does for the mergesort-based version.

One might wonder whether this suffix array generation problem could be solved efficiently by direct application of a known parallel quicksort algorithm [8]. The answer is no for a couple of reasons. First, the generation of a suffix array is an indirect sorting procedure because the sorting is done on the index pointers. Second, suffix arrays generation require comparing large (pruned) sistrings which need to be accessed remotely and thus, extra care must be exercised.

The quicksort-based algorithm considers the presence of r processors connected to a high-bandwidth switching device. Each processor has a local memory (we assume for simplicity that all memories are of the same size, although it is possible to cope with different-size memories) which stores a piece of the text. Each piece of text is called a *block*. The number of words in a block is referred to by b. Further, the number of words in the whole text is referred to by n. Thus,

$$n = r * b \tag{1}$$

Our algorithm starts by determining the beginning of each sistring in the text (i.e., the beginning of each word) and by generating the corresponding index pointer. This task is done in parallel for each of the r blocks of text. After this computation, each local memory contains an array composed of b index pointers. Since computation of the whole suffix array requires moving index pointers among processors without losing sight of the sistrings they point to, index pointers are computed relative to the whole text.

Each processor proceeds by executing the following steps:

(1) compute the lexicographical median sistring for the local block of text;
(2) broadcast this median to the other $r - 1$ processors;
(3) compute the median m of the r local medians

The processors then exchange index pointers among them such that each processor has all its index pointers pointing to sistrings which lie either to the right or to the left of the computed median m. Once this step is concluded, we are left with two subproblems which are identical to the original problem but smaller. The algorithm is then applied recursively to each of these two subproblems (as the quicksort). If the computed median m approximates the real media well, the two subproblems are roughly the same size and the algorithm performance is good. In the Appendix we show that this is the expected case and that the estimation of the median improves as the number r of processors increases.

Exchanging index pointers leaves us with a problem: a given processor i now stores index pointers to sistrings which reside in the local memory of another processor. This means that to be able to proceed with the partitioning of the index pointer arrays the processors must access sistrings stored remotely. Since a single access to a remote (pruned) sistring requires two full ATM packets (one for requesting the pruned sistring and other for retrieving it), a lot of bandwidth is wasted. Fortunately, such drawback can be avoided by moving (from one processor to another) not only the index pointer but also its corresponding pruned sistring.

However, care must be exercised because (pruned) sistrings are long (48 bytes) and present a lot of overlapping (i.e., sistrings which begin at consecutive words in the text overlap almost entirely) which makes moving entire sistrings prohibitive. The key insight is to move only the first ℓ characters of each sistring and use these ℓ characters to decide comparisons of sistrings. In a random text composed of words formed by σ distinct characters, the probability that the first 6 characters of a sistring will not be enough for deciding a comparison is given by $\frac{1}{\sigma^\ell}$.

In a natural language text, however, words overlap more frequently. Consider for instance a large collection of texts such as the Tipster collection [5] (which contains articles extracted from different fonts including the Wall Street Journal). The Tipster collection includes over half a million documents. Empirical evidence collected from this collection (see section) suggests that a reasonable value for ℓ is 10. Such value for ℓ allows deciding about 85% of all comparisons. The remaining 15% could be decided by accessing the complete (pruned) sistrings remotely. A special pivoting process can be adopted to avoid the occurrence of such operation (remote access to the full sistring) at the first levels of our quicksort-based algorithm. The idea is as follows.

In the pivoting process, a pruned sistring which is found to be equal to the (pruned) pivot is put at the left partition. This works well and avoids at all requesting full sistrings to other processors. However, as the algorithm progresses, this pivoting process can worsen the randomness of the partition. Further, such effect tends to get worse at the final stages of the sorting process. At some point, full sistrings need to be requested remotely to allow generation of the final sorting. The number of such remote accesses depends on the text and on the size of the local memories. Refer to section 4 for further details.

We now describe the algorithm more formally. The working mode of our algorithm reduces the problem of generating the suffix array to one of an indirect sorting in a distributed memory. Let $E(i)$ be the set of index pointers stored in the processor i. Further, let p be a reference to an index pointer and let $S(p)$ be the pruned sistring pointed to by p. The algorithm performs a number of recursive iterations as follows.

1. Each processor i executes the following steps:

   (a) compute the median $m(i)$ for its block of text;
   (b) broadcast the median $m(i)$;
   (c) knowing all the other medians, compute $m = median\{m(1), ..., m(r)\}$;
   (d) partition its index pointers in two sets $L(i)$ and $R(i)$:

   $$L(i) = \{p \in E(i) | S(p) \leq m\}; \quad R(i) = \{p \in E(i) | S(p) > m\} \tag{2}$$

   (e) broadcast the sizes $|L(i)|$ and $|R(i)|$ of the computed partitions.

2. The processors engage in a *redistribution* process in which they exchange index pointers until each processor contains all of its index pointers in either L or R where

   $$L = \bigcup L(i); \quad R = \bigcup R(i) \tag{3}$$

   We say that the processor becomes *homogeneous* when this happens. There can be left at most one processor whose index pointers lie in both L and R (we explain later how this is accomplished). This processor is called *non-homogeneous*.

3. This redistribution of index pointers splits the processors in two groups: those whose index pointers belong to L and those whose index pointers belong to R. The two groups of processors proceed independently and apply the algorithm recursively.

4. The recursion ends whenever an L or R set of index pointers lies entirely in the local memory of a processor. In this case, all that remains to be done is to sort L or R in the local memory.

The process of redistributing index pointers is carried out in a number of steps which are completely planned inside each processor (simulating completion times for exchanges) and later followed independently. This avoids exchanging synchronization messages. To accomplish such effect, the processors are paired in a fixed fashion (for instance, pair the processor $(2i)$ with the processor $(2i+1)$ for all i). Each pair manages to exchange a minimum number of index pointers such that one of them is left homogeneous. The homogeneous processor in each pair is left outside of the redistribution process. The remaining half processors engage in a new redistribution process in which the processor $(4i)$ or $(4i+1)$ is paired with the processor $(4i+2)$ or $(4i+3)$ (depending on which one is still non-homogeneous). Note that, since all processors have the information needed to predict the redistribution process, they know which processor to pair with at each iteration. This ends when there is only one non-homogeneous processor.

Let us focus in the task of making one of the processors in a pair homogeneous. Consider the pair composed of processors a and b. By comparing its sistrings with the computed median m, the processor a separates its index pointers according to the internal partition (L_a, R_a). Analogously, the processor b separates its index pointers according to the internal partition (L_b, R_b). Let $|L_a|$,$|R_a|$,$|L_b|$, and $|R_b|$ be the number of index pointers in each of these partitions. Without loss of generality, let $\min(|L_a|, |R_a|, |L_b|, |R_b|) = |L_a|$. Then, processor a can make himself homogeneous by sending all the index pointers in its partition L_a to processor b while retrieving (from processor b) $|L_a|$ index pointers of partition R_b. After this exchange, processor a is left with all its index pointers belonging to R (and thus, homogeneous) while processor b is left with a partition $(L_b \bigcup L_a, R'_b)$, where $R'_b \subset R_b$ and $|R'_b| = |R_b| - |L_a|$. The other cases are analogous.

The non-homogeneous processor could potentially slow down the process, since it has to act in two (parallel) groups. Although it does not affect the total complexity (since a processor belongs at most to two groups), it can affect the constants. To alleviate the problem, we can mark it so that in the next redistribution process it is made homogeneous in the first exchange iteration. It may take longer, but the processor is free for the rest of the iterations.

5 Analysis of the Quicksort-Based Algorithm

We analyze the behavior of our algorithm in the worst and average cases. The analysis accounts for both internal and communication costs, although we later concentrate on the second ones.

5.1 Worst Case

We consider the cost $T(r)$ of our distributed algorithm described in section 4. Since the size of the problem is reduced at each recursion step, the number of processors in the newly generated L and R groups decreases. We consider the cost of a recursive step with r processors initially. The final part of this cost is that of solving the subproblems it generates. We account for internal processing costs with a factor $\mathbf{I}$, and communication costs with $\mathbf{C}$. We do not count a communication cost additionally as an internal cost.

The cost $T(r)$ of our algorithm for r processors is as follows:

1. Costs for each processor i (costs are parallel for all processors):
   (a) computation of the median $m(i)$ is $O(b)$ $\mathbf{I}$;
   (b) broadcasting the median $m(i)$ is $O(r)$ $\mathbf{C}$;
   (c) computation of the median m is $O(r)$ $\mathbf{I}$;
   (d) partitioning of index pointers in sets $L(i)$ and $R(i)$ is $O(b)$ $\mathbf{I}$;
   (e) broadcasting the sizes $|L(i)|$ and $|R(i)|$ is $O(r)$ $\mathbf{C}$.

2. Cost of redistributing index pointers in subproblems L and R is as follows. There are $\log_2 r$ steps because half of the processors is made homogeneous at each redistribution step. Since at most b index pointers are exchanged in each step (because $\min(|L_a|, |R_a|, |L_b|, |R_b|) \leq b/2$), the total cost is $O(b \log_2 r)$ $\mathbf{C}$.

3. Cost for the recursive calls (processing of groups L and R) depends on the worst-case partition. Let r_L be the number of processors in group L and r_R be the number of processors in group R. We show that $r/4 \leq r_L \leq 3r/4$ in the worst case. Hence, there are at most $\log_{4/3} r$ levels in the recursion in the worst case. The number of processors in the larger partition is at most $3/4r$ (the smaller partition works in parallel and does not affect completion times). Therefore, $T(3/4r)$ must be added to $T(r)$.

4. Cost of sorting the index pointers locally is $O(b \log b)$ $\mathbf{I}$.

To show that $r/4 \leq r_L \leq 3r/4$, observe that the estimated median m is larger than $r/2$ local medians, each one is in turn larger than $b/2$ elements of the corresponding processor. Hence, m is larger than $n/4$ elements which implies that r_L is larger than $r/4$. The demonstration for the upper bound is analogous.

The complexity of the total execution time is given by the recurrence

$$T(1) = O(b \log b)\ \mathbf{I}$$
$$T(r) = O(b + r)\ \mathbf{I} + O(r + b \log r)\ \mathbf{C} + T(3/4r)$$

which gives

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

where we can assume $r < b$ to obtain $T(r) = O(b \log n)\ \mathbf{I} + O(b \log^2 r)\ \mathbf{C}$.

Observe that the internal part is almost optimal, since the time using r processors is almost $1/r$ times that of a single processor. Hence, we concentrate in communication costs. The exact constants for the main part of the cost lead to $b \log_2 r \log_{4/3} r$.

Consider the optimal execution of our algorithm. Such optimal execution occurs when all local medians coincide with the global median (the optimal execution might also occur with less strict restrictions but this does not affect our analysis). In this situation, the cost of redistributing index pointers in subproblems R and L is $O(b)$. Therefore, the overall cost of an optimal execution of our algorithm is $O(b \log r)$ which is about an order of magnitude smaller than our estimated cost of $O(b \log^2 r)$.

One main problem with our previous mergesort-based distributed algorithm [9, 6] is scalability. In the mergesort algorithm, doubling both the size of the text and the number of available processors results in an increase of the execution time by a factor of 2. Since one should expect the execution time to remain constant, the algorithm does not scale well. Let us analyze what happens with the quicksort-based algorithm presented here. If we double n and r, the new communication cost $C(2n, 2r)$ becomes

$$C(2n, 2r) = C(n, r)\left(1 + \frac{2}{\log_2 r}\right)$$

where the ideal scalability condition is $C(2n, 2r) = C(n, r)$. While our quicksort-based algorithm does not scale ideally, it does scale much better than the previous mergesort-based algorithm. Further, as the number of processors increase, the additional computational time (given by the fraction $2/\log_2 r$) drops considerably. For instance, if the number of processors doubles from 256 to 512 the execution time goes up by a factor of 25% (instead of also doubling).

5.2 Average Case

We prove in this section that the algorithm works almost optimally in the average case. The main part of the proof is to show that, for large n, the estimated median is almost the exact median. Once we prove that, we have that each redistribution session exchanges almost all the data in a single step (since $|L_a| \approx |L_b| \approx |R_a| \approx |R_b|$), being the remaining steps so small (in terms of communication amounts) that can be safely ignored. Moreover, since the partition is almost perfect, $|L| \approx |R|$, and the next subproblems are half the size of the original one. This makes the total communication cost $O(b \log r)$ on average (or $b \log_2 r$ for the exact constant), and the network is being used all the time. The scalability factor of this version is $1 + 1/\log_2 r$.

The non-homogeneous processor does not add too much to the cost, since it has $\approx b/2$ elements in each partition, and hence exchanges $\approx b/4$ on each group. This takes the same as exchanging $b/2$, which is the normal case in the other processors.

To prove that the partition is almost exact, we consider every local median as an estimate of the global median obtained by taking b random samples out of n. We show that this estimate is very close to the real median. Since the global estimate is obtained by taking the median over r local estimates, its distribution is even better. The proof is quite involved, and we leave it to the Appendix.

6 Conclusions and Future Work

We have discussed a quicksort-based distributed algorithm for the generation of suffix arrays for large texts. The algorithm is executed on workstations connected through a high-bandwidth network. The key idea is to use the aggregate memory of the various processors as a giant cache for disks. In such aggregate memory, remote accesses are as time consuming as sequential accesses to a local disk. Appart from parallelism, the improvement in performance comes from the fact that remote memory accesses can be done randomly.

We analyzed the average and worst case complexity of our algorithm considering a text of size n and the presence of r processors. Such analysis demonstrates that the algorithm presents two important advantages over a previous parallel algorithm based on the mergesort. First, the quicksort-based algorithm has a running time whose complexity is much lower ($O(n)$ for the mergesort-based algorithm and $O(n \log_2 r \ / \ r)$ for the quicksort-based algorithm). Second, the quicksort-based algorithm scales up much nicer than the mergesort-based algorithm. For instance, consider operation with 256 processors. If one doubles the size (n) of the problem and also doubles the number of processors, the running time of our quicksort-based algorithm increases by 25% while the running time for the mergesort-based algorithm doubles.

We are currently working on the implementation of both the mergesort and the quicksort based algorithms. The mergesort implementation is basically concluded and we are comparing its performance with that of a local implementation of the sequential algorithm. Soon, we expect to be able to also collect empirical data on the performance of the quicksort based algorithm. Besides such implementation efforts, we are investigating the application of our ideas to the generation of the more popular inverted lists.

References

[1] Thomas Anderson, David Culler, and David Patterson. A case for NOW (network of workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[2] Jeffrey Kuskin et all. The stanford flash multiprocessor. In *21st International Symposium on Computer Architecture*, pages 302–313, Chicago, April 1994.

[3] Gaston Gonnet. *PAT 3.1: An Efficient Text Searching System – User's Manual.* Centre of the New Oxford English Dictionary, University of Waterloo, Canada, 1987.

[4] Gaston H. Gonnet, Ricardo A. Baeza-Yates, and Tim Snider. New indices for text: Pat trees and pat arrays. In *Information Retrieval – Data Structures & Algorithms*, pages 66–82. Prentice Hall, 1992.

[5] Donna Harman. Overview of the third text retrieval conference. In *Proceedings of the Third Text Retrieval Conference - TREC-3*, Gaithersburg, Maryland, 1995. National Institute of Standards and Technology. NIST Special Publication 500-225.

[6] João Paulo Kitajima, Berthier Ribeiro, and Nivio Ziviani. Network and memory analysis in distributed parallel generation of pat arrays. In *Fourteenth Brazilian Symposium on Computer Architecture*, Recife, August 1996.

[7] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. In *First ACM-SIAM Symposium on Discrete Algorithms*, pages 319–327, San Francisco, 1990.

[8] Michael J. Quinn. *Parallel Computing: Theory and Practice*. McGraw-Hill, second edition, 1994.

[9] Berthier Ribeiro, João Paulo Kitajima, and Nivio Ziviani. Distributed parallel generation of pat arrays. Technical Report 019/96, Universidade Federal de Minas Gerais - Departamento de Ciência da Computação, Belo Horizonte, Brazil, June 1996. In English.

Appendix: Average Median by Sampling b Out of n

For simplicity, we assume that $b = 2m + 1$. The probability of our estimated median being the position j in the sorted array is that of, in our sampling, selecting m elements in the range $[1..j-1]$, m elements in the range $[j+1..n]$, and of course selecting j. This is

$$s(j) = \frac{\binom{j-1}{m}\binom{n-j}{m}}{\binom{n}{2m+1}}$$

We are interested in the expected proportional size of the larger partition. This is

$$P = \frac{1}{n}\left(\sum_{j=1}^{n/2}(n-j+1)s(j) + \sum_{j=n/2+1}^{n} js(j)\right) = 2\sum_{j=n/2+1}^{n}(j/n)s(j)$$

We call $t(j) = js(j)$ and $f(x) = t(nx)$ the continuous version of $t(j)$ over the interval $[1/2 .. 1]$. Hence

$$P = \frac{2}{n}\sum_{j=n/2+1}^{n} t(j) = \frac{2}{n}\sum_{j=n/2+1}^{n} f(j/n) \leq 2\int_{1/2}^{1-m/n} f(x)dx$$

(since $t(j)$ is descending for large n).

Since $f(x) = t(nx)$, it follows that

$$\frac{f(x+1/n)}{f(x)} = \frac{t(nx+1)}{t(nx)} = v(nx)$$

where we have just defined

$$v(j) = \frac{t(j+1)}{t(j)} = \frac{(j+1)(n-m-j)}{(j-m)(n-j)}$$

Taking logarithms, we have

$$\ln f(x+1/n) - \ln f(x) = \ln v(nx)$$

$$\frac{\ln f(x+1/n) - \ln f(x)}{1/n} = n \ln v(nx)$$

This last equation defines $(\ln f)'$, hence

$$f(x) = K\ e^{n\int_{1/2}^{x} \ln v(ny)dy}$$

the constant K coming from the integration. We obtain it observing that $f(1/2) = K = t(n/2)$, from where

$$f(x) = \sqrt{\frac{m}{\pi}}\ e^{n\int_{1/2}^{x} \ln v(ny)dy}(1 + O(m/n) + O(1/m))$$

We now solve the integral of $\ln v(ny)$. We have

$$\begin{aligned} n\int_{1/2}^{x} \ln v(ny)dy &= \int_{n/2}^{nx} \ln v(z)dz \\ &\leq m(2\ln 2 + \ln x + \ln(1-x)) + \ln 2 + \ln x + O(1/n) \end{aligned}$$

We then rewrite the equation for $f(x)$ as follows

$$f(x) = \sqrt{\frac{m}{\pi}}\, 2^{2m+1} x^{m+1}(1-x)^m (1 + O(m/n) + O(1/m))$$

and return to our wanted result on P

$$P \leq 2\int_{1/2}^{1-m/n} f(x)dx \quad \leq \quad \frac{4^{m+1}\sqrt{m}}{\sqrt{\pi}} \int_{1/2}^{1} x^{m+1}(1-x)^m dx \ (1 + O(m/n) + O(1/m))$$

This last integral is not trivial. We solve it by induction. Let

$$h(d) = \int_{1/2}^{1} x^{m+1+d}(1-x)^{m-d}dx$$

then

$$h(m) = \frac{1 - \frac{1}{4^{m+1}}}{2m+2}$$

and our desired result is $h(0)$. Using $fg = \int f'g \ + \ \int fg'$, we have

$$h(d) = \int_{1/2}^{1} x^{m+1+d}(1-x)^{m-d}dx = \frac{1}{(m-d+1)4^{m+1}} + \frac{m+d+1}{m-d+1}\int_{1/2}^{1} x^{m+d}(1-x)^{m-d+1}dx$$

where the last integral is $h(d-1)$, hence the recurrence. By using $g(i) = h(m-i)$ we have the more conventional one

$$g(0) = \frac{1 - \frac{1}{4^{m+1}}}{2m+2} \quad , \quad g(i+1) = \frac{(i+1)g(i) - \frac{1}{4^{m+1}}}{2m-i+1}$$

which yields

$$g(m) = \frac{1}{8}\frac{\sqrt{\pi/m}}{4^m}\ (1 + O(1/m))$$

and hence we have the final result

$$P \leq \frac{1}{2}\ (1 + O(m/n) + O(1/m)) = \frac{1}{2}\ (1 + O(b/n) + O(1/b))$$

what shows that the estimated median is very close to the real median for moderately large b.

A question that naturally arises is why the result seems to be worse as b grows (i.e. the $O(b/n)$ error term). This is because we used upper bounds in some parts, hiding factors depending on m that made the result smaller. Since it is clear that, as b grows, the estimation gets better, and that we can assume $b > r$ (i.e. $b > \sqrt{n}$), we have an estimation error independent of b

$$P \leq \frac{1}{2}\ \left(1 + O\left(1/\sqrt{n}\right)\right)$$