Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Distributed Parallel Generation of Pat Arrays

Berthier A.N. Ribeiro
João Paulo W. Kitajima
Nivio Ziviani

Relatório Técnico 019/96[1]

June 1996

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

Distributed Parallel Generation of Pat Arrays

Berthier A.N. Ribeiro

João Paulo W. Kitajima

Nivio Ziviani†

Computer Science Department
Federal University of Minas Gerais
Belo Horizonte, Minas Gerais, Brazil
e-mail: {berthier,kitajima,nivio}@dcc.ufmg.br

Abstract

We propose a parallel and distributed algorithm for generation of large pat arrays in the environment of a high-bandwidth network of workstations. As starting point we consider a well known sequential algorithm which is quite efficient when operating on a single machine. We show, however, that its inherently sequential nature prevents large improvements in execution time when the algorithm is modified to run on a network of workstations. We then demonstrate that our distributed parallel algorithm improves the execution time of the sequential algorithm by a factor of N_b, where N_b is the number of blocks of partial text. Such improvement makes the algorithm linear on the size of the text (i.e., $O(N_b m)$ where m is the amount of available memory in each machine) and provides speedups which can go up to one order of magnitude and beyond when commercially available workstations are considered. This is accomplished by carefully exploring the aggregate memory of the workstations in the network.

1 Introduction

Representing the content of a document and of a user query by a set of keywords was necessary in the early seventies due to performance constraints. Nowadays, the advent of powerful workstations has allowed the consideration of alternative models for information retrieval. One such 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[1]. The user expresses his information need by providing strings to be matched and the information system retrieves those documents which contain the user specified strings. The cost of searching the full text is usually high but the method presents important advantages [3]. First, no structure in the text is needed which broadens the scope of applications of full text search. Second, no keywords are used which broadens the domain of queries the user might specify. Third, the model is simpler and can be easily grasped by a common user.

†This author has been partially supported by CNPq Project 520916/94-8 and Project RITOS/CYTED.

[1]The 2 gigabytes TIPSTER collection [4] includes thousands of documents which are articles extracted from editions of the Wall Street Journal, AP Newswire, abstracts from DOE publications, and US Patents among others.

The large size of a full text collection (consider, for instance, that the texts of all documents in the collection are merged in a single very large text file[2]) and the potentially large size of the string to be searched for demand specialized index techniques for efficient retrieval. A fast index for retrieval in full text is the Patricia tree [9]. Indeed, when long search strings are involved the Patricia tree is the method of choice [5, 10]. Pat arrays [2] or suffix arrays [7] are simpler structures which provide an efficient implementation of Patricia trees while supporting a powerful query language.

The purpose of this paper is to investigate algorithms for the generation of large pat arrays in the context of a high-bandwidth network of workstations. A network of workstations is an attractive alternative nowadays due to the following reasons. First, the emergent fast switching technology provides very fast message exchanges among the various machines and consequently less parallelism overhead. Second, a network of workstations provides computing power comparable to that of a typical parallel machine but is more cost effective [1]. Third, these workstations are already available elsewhere and most of them are completely idle most of the time [1].

The emergent ATM technology provides fast message exchanges (155 Mbps, megabits per second, is a commercial reality these days) between any pair of nodes (without contention) which is a requirement for efficient parallelism. However, this is not enough. Besides low latency in the network, 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 in a conventional workstation one has to deal with the delays imposed by the operating system. The user must transmit directly into and receive from the network without interference from the operating system. This can be attained by mapping data directed to the network interface into the user address space directly. 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, 6]. The NOW project [1] aims at performing user-to-user communication of a small message (48 to 192 bytes) among 100 machines in 10μs. Such goal is technologically feasible and should become a reality in a few years.

Fast message exchanges make it attractive to use the aggregate memory in the high-bandwidth network as a giant cache for disks. Such alternative is viable because even during the daytime hours, more than 60 percent of the workstations in a local network are available 100 percent of the time [1]. Our idea is to utilize the aggregate memory in the network as storage for the whole text and its associated pat array.

Efficient execution in this context requires the algorithm to carefully balance the effects of message communication delays, disk access times, memory access times, and memory utilization while attempting to achieve the highest degree of parallelism. Further, the algorithm must also explore the potential parallelism between accesses to the aggregate memory and accesses to the large number of disks (which can perform seek operations in parallel) distributed over the network. As very properly stated in [10]: *that a clever method might go unappreciated or unused because of a simple change in the technology is a definite possibility in external sorting.*

In this paper we investigate the performance of three algorithms for generating large pat arrays in a high-bandwidth network of workstations. The three algorithms are: (1) the sequential algorithm presented in [3] which we refer to as *Patseq*, (2) a parallel version for

[2] As done with the TIPSTER collection [4].

it which we refer to as *Patseqpar*, and (3) a distributed parallel mergesort algorithm which we refer to as *Patpar*. We demonstrate that the inherently sequential nature of the *Patseq* algorithm prevents its parallel counterpart (i.e., the *Patseqpar* algorithm) from yielding large improvements in execution time. We also demonstrate that the Patpar algorithm provides an improvement in execution time which might go up to one order of magnitude and beyond when commercially available workstations are considered. Such performance figure is not easily attainable though and requires careful design.

2 Basic Definitions

In this section we introduce basic definitions and terminology used throughout the paper. We also discuss basic characteristics of the pat arrays we generate.

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 k and extends to the right as far as needed or to the end of the text. Such semi-infinite strings are called *sistrings* [3]. 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 user specified sistrings. To perform this search efficiently our information system uses a pat array (generated for that text) as the indexing structure.

A *pat array* is a linear structure composed of pointers to every sistring in the text. These pointers are sorted according to a *lexicographical ordering* of their respective sistrings. Further, each of these pointers can be viewed simply as the offset (counted from the beginning of the text) of the sistring in the text.

We consider that the user is interested in text words and/or text phrases only such that the pat array needs to include solely pointers to those sistrings which occur at the beginning of words in the text (less than 25% of a common text written in English) [3]. Let T be the size of the text. Assuming that the pointers in the pat array are words of 4 bytes, the size of the pat array is approximately T (i.e., $4 \times 0.25 \times T$). Thus, the pat array and the text are roughly the same size.

Searching for a sistring might require a potentially large number of character comparisons because of the sistring semi-infinite property. However, the vast majority of expected search requests do not require comparing very large sistrings. For instance, searching the Oxford English Dictionary for user specified sistrings can be resolved in 97% of the cases with comparisons which involve no more than 48 characters [3]. Thus, in the rest of this paper we consider that sistrings are 48 characters long.

3 The *Patseq* and *Patseqpar* Algorithms

This section discusses the algorithm for generation of pat arrays presented in [3] (which is called here *Patseq*) and a parallel version for it called *Patseqpar*.

3.1 The Sequential Version – *Patseq*

The *Patseq* algorithm considers that a single machine is used to build the pat array. This machine must have enough disk space for storing the whole text and its pat array. For instance, the disk space required for storing a text of size 1 gigabyte and its pat array is roughly 2 gigabytes (for a text containing 25% of sistrings).

If all the text and its pat array fit in main memory, the problem becomes one of sorting the pat array (according to a lexicographical ordering of its corresponding sistrings) in primary memory and can be solved trivially using (for instance) quicksort. For large texts, however, the text must reside on disk and one cannot avoid accessing the secondary memory for retrieving text sistrings. Since accesses to secondary memory are dominated by the seek time, one must minimize the number of seek operations when accessing a large file. This is the main concern of algorithms which perform *external sorting*.

Most external sorting algorithms are based on the following procedure [5, 10]:

Algorithm 1 *General procedure for external sorting.*

(1) break the large file into smaller blocks which fit in main memory;
(2) sort the individual blocks in main memory;
(3) merge the sorted blocks together until the whole file is sorted.

The key advantage of this procedure is that data are accessed sequentially most of the time. Indeed, the *Patseq* algorithm is based on it.

Let T be the length of the whole text in bytes and M, $M \ll T$, be the memory available for generation of the pat array. We break up the text in blocks of size B bytes such that $3 \times B = M$. The reason for the constant 3 will become clear ahead. Let N_b be the number of blocks contained in the full text (i.e., $N_b = \lceil \frac{T}{B} \rceil$). Once the number N_b of blocks has been decided, the step (2) of Algorithm 1 can be executed as follows. For each block b_i do: (a) load the block b_i sequentially into main memory; (b) using quicksort, generate (in main memory) the pat array p_i corresponding to the block b_i; and (c) store the pat array p_i sequentially on disk (this phase corresponds to the generation of small pat arrays – one for each text block). The phases (a) and (c) require a single seek operation at the beginning. After this seek operation, the access is purely sequential. After step (2) of Algorithm 1 is concluded, the merging of the N_b pat arrays (step (3)) is initiated. Unfortunately, such merging operation is more involved than it seems at first glance.

A very simple (and naïve) approach to merge the N_b pat arrays is as follows (specified in a C-like language):

Algorithm 2 *Naïve algorithm for merging partial pat arrays.*

$P = p_1$; / P is the resultant pat array */*
/ merge the small pats with the large partial pat which is already sorted */*
for ($i = 2$; $i \leq N_b$; $i++$) $P = merge(P, p_i)$;

The key problem is the implementation of the function called *merge*. Let b_P be the portion of the text (composed of one or more blocks of text) corresponding to the pat array P. To merge the pat arrays P and p_i, it is necessary to access the sistrings in b_P and b_i. Since these accesses are non-sequential (they follow the pat array pointers), the text blocks b_P and b_i must be in main memory (otherwise, too many seek operations would be required). Unfortunately, as P grows b_P becomes too large to fit in main memory. If we attempt to access the sistrings in b_P directly from disk, the execution time becomes unacceptable due to the large number of seek operations.

To avoid the seek operations, the *Patseq* algorithm always accesses the text sequentially (i.e., subsequent disk accesses refer to contiguous disk sectors). This is accomplished by maintaining auxiliary counters which indicate the number of sistrings, pointed by elements of a pat, which fall between any two consecutive sistrings pointed by elements of another pat in a lexicographical ordering of the sistrings. These counters allow merging the two pat arrays through strictly sequential accesses (with no need to further inspect text sistrings). Most important, these counters can be generated by always accessing text sistrings sequentially. Thus, the whole process can be completed through strictly sequential disk accesses. Counters generation is done as follows.

Consider two pat arrays p_i and p_j and let n_j be the number of elements in p_j. The counters which indicate how to intercalate p_i into p_j are maintained in an array $C_{i,j}$ containing exactly $n_j + 1$ integers. Let $p_j[k]$ be a reference to the kth pointer in the pat array p_j. The element $C_{i,j}[k]$ counts the number of sistrings reachable from p_i which lie between the sistring pointed by $p_j[k]$ and the sistring pointed by $p_j[k+1]$ (assume the presence of two sentinels $p_j[0]$ and $p_j[n_j+1]$). To generate $C_{i,j}$ the following three data structures are kept in main memory: the pat array p_j, its block of text b_j, and the array $C_{i,j}$ itself (this is the reason for dividing the primary memory in blocks of size $\frac{M}{3}$). The sistrings pointed by the pat p_i (i.e., the text block b_i), $i < j$, are then retrieved sequentially from disk and processed as follows.

Algorithm 3 *Computing the array of counters $C_{i,j}$.*

foreach "sistring $\mathcal{S}, \mathcal{S} \in b_i$, retrieved sequentially from disk" do {
k = "position of $\mathcal{S}$ in p_j";
$C_{i,j}[k]{+}{+}$; / increment the counter */*
}

The *"position of $\mathcal{S}$ in p_j"* is determined through an indirect binary search in the pat array p_j. This search reveals a pair of pointers $p_j[k]$ and $p_j[k+1]$ which point to the two sistrings of b_j surrounding $\mathcal{S}$ ($\mathcal{S}$ is in b_i) in a lexicographical ordering of them. The counter $C_{i,j}[k]$ is incremented to reflect this fact.

Figure 1 illustrates the data structures used by the *Patseq* algorithm for merging the N_b pat arrays. The merging is done as follows. Firstly, the array of counters $C_{1,2}$ is computed

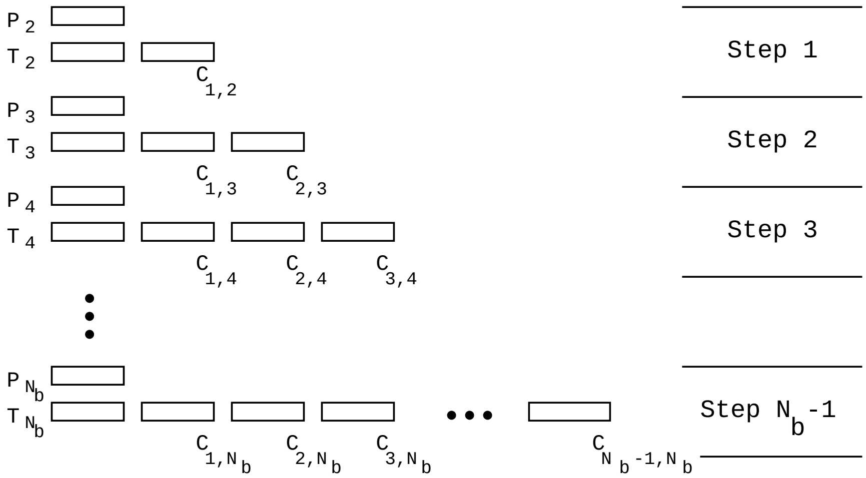

Figure 1: Data structures used by the *Patseq* algorithm for merging the N_b pat arrays.

and stored on disk. This array allows merging the pat array p_1 into the pat array p_2 to generate the resultant pat array p_{1+2}. Secondly, the arrays $C_{1,3}$ and $C_{2,3}$ are computed and summed up on the fly to yield the resultant array of counters $C_{1+2,3}$ which is stored on disk. This resultant array allows merging the composed pat array p_{1+2} into the pat array p_3. In the jth step, the array $C_{1+\ldots+(j-1),j}$ is computed and stored on disk. Once all counters have been computed, the algorithm is ready to conclude the merging of the pat arrays which is accomplished as follows.

Algorithm 4 *Using the arrays of counters to merge the partial pat arrays.*

$P = p_1;$

for $(j = 2;\ j \leq N_b;\ j++)$ *merge_using_counter*$(P,\ p_j,\ C_{1+\ldots+(j-1),j});$

The procedure *merge_using_counter* simply accesses the two pat arrays sequentially and merges them guided by the array of counters. Since P becomes larger and larger it is kept on disk. That is not a problem because P is always accessed sequentially.

3.2 The Parallel Version – *Patseqpar*

The *Patseq* sequential algorithm is composed of two main tasks: (1) task *counter_computing* which generates the counters and (2) task *pat_merging* which produces the resultant pat $p_{1+\ldots+N_b}$. Unfortunately, this merging task has an inherently sequential nature. The computation of the partial pat p_{1+2+3} can only be initiated *after* the pat p_{1+2} has been computed. In general, computation of $p_{1+\ldots+j}$ can only be done after $p_{1+\ldots+(j-1)}$ has been computed. Since the whole pat_merging task has a complexity given by $O(N_b^2 M)$ (see Subsection 3.3.2), this is the complexity of any parallel version of the *Patseq* algorithm. Thus, no matter how many machines are available, the complexity of *Patseq* cannot be improved. However, the expected execution time can be improved by parallelizing the counter_computing task.

In [3] it is suggested a parallelization of the counter_computing task which assigns the computation of each array of counters $C_{i,j}$ to a distinct processor. While this approach might work well with a parallel machine, it is not practical in the environment of our high-bandwidth network for two reasons. First, it requires too many processors which are not available in a network of workstations. For instance, for a text of size 1 Gbyte and machines with 64 Mbytes of available memory the number N_b of text blocks is 48. The number of counters is then $\frac{N_b * (N_b - 1)}{2}$ and thus, 1128 workstations would be required. Second, too many messages would have to be exchanged across the network to accumulate the counters ($C_{1+\ldots+(j-1),j} = C_{1,j} + C_{2,j} + \ldots + C_{j-1,j}$). For our purposes, a different approach is required.

We consider that our high-bandwidth network contains at least N_b machines available and that each of them has a free space of size M bytes in its primary memory (non-virtual). Furthermore, we assume that there is disk space available in each machine's local disk. The key insight for a good parallelization of *Patseq* is noticing that the counter_computing and the pat_merging tasks can be run in parallel. This is accomplished by (1) restricting the counter_computing task to retrieve data from local disks and (2) restricting the pat_merging task to retrieve data from the aggregate memory. The result is that the data access patterns of the two tasks do not interfere with each other.

Given the above considerations, *Patseqpar*, our parallel implementation of *Patseq*, comes up very naturally. The blocks of text are replicated in the local disks of the various machines as follows: machine 1 stores the block of text b_1, machine 2 stores the blocks of text b_1 and b_2, and so on. Following, the N_b machines compute, in parallel, the initial pat arrays p_1, p_2,

..., p_{N_b}. The machine i is in charge of computing p_i which it accomplishes as follows: (a) read b_i from disk and (b) compute p_i using quicksort. After this step, the machine i holds in its main memory the pat p_i and its corresponding text block b_i.

To compute the counters, the algorithm associates to each step i in Figure 1 processor $i+1$. Thus, processor 2 is responsible for computing $C_{1,2}$, processor 3 is responsible for computing $C_{1+2,3}$, and so on. The computed counters are kept in main memory. During this phase, all processors execute Algorithm 3 (which requires only local disk accesses) and operate fully in parallel. As soon as $C_{1,2}$ is computed, processor 2 can initiate the pat_merging task (there is no need to wait for the other processors). It does so by retrieving the pointers of p_1 (across the network, from the memory of the machine 1) and by merging them into p_2 (guided by $C_{1,2}$) to generate p_{1+2}. Since $C_{1,2}$ and p_2 reside locally in the memory of the processor 2, no disk accesses are made. The resultant pat p_{1+2} is stored in the aggregate memory of the machines 1 and 2 (in the space previously occupied by the text block b_1 in machine 1 and the text block b_2 in machine 2). In general, after computing the counter $C_{1+\ldots+(j-1),j}$ the machine j needs to wait only for the computation of $p_{1+\ldots+(j-1)}$ by the machine $j-1$ to enter the pat_merging task. The overall effect is that the counter_computing and the task_merging tasks march in parallel.

A drawback of this approach is that it requires extra space in disk which is proportional to $\frac{N_b*(N_b-1)}{2} * B$. For $T = 1$ Gbyte, $M = 64$ Mbytes, and $N_b = 48$, approximately 24 Gbytes of extra disk space is required.

3.3 Modeling the Execution Times

In this section we model the execution times for the *Patseq* and the *Patseqpar* algorithms. The model takes into account physical parameters such as memory size, memory bandwidth, disk bandwidth, network bandwidth, etc. Such detailed consideration is important to elucidate critical trade-offs which are difficult to grasp in a conventional study of complexity.

3.3.1 Physical Parameters

According to our previous considerations (Section 2), text blocks, pat arrays, and counter arrays are roughly the same size. Further, T, M, and N_b stand for the full text size, the memory available in each machine, and the number of text blocks, respectively. Clearly,

$$N_b = \frac{T}{\frac{M}{3}}$$

Let,

bw_{mem}: memory bandwidth in bytes per second
bw_{dsk}: disk bandwidth in bytes per second
$t_{wordmem}$: time to retrieve a 4 bytes word from memory $(4 \times \frac{1}{bw_{mem}})$
$t_{worddsk}$: time to retrieve a 4 bytes word from disk $(4 \times \frac{1}{bw_{dsk}})$
s_{blk}: size of a text block in bytes $(= \frac{M}{3})$
s_{pkt}: size of a network packet in bytes (= 48, ATM technology)
t_{pkt}: time in seconds to move a user-to-user packet from one machine to another
t_{net}: time in seconds to move a text block from one machine to another
t_{dsk}: time in seconds to read/write a text block from a local disk
t_{mem}: time in seconds to read/write a text block from local memory

Then,

$$t_{net} = \frac{s_{blk}}{4} \times \frac{4 * t_{pkt}}{s_{pkt}}; \qquad t_{dsk} = \frac{s_{blk}}{4} \times t_{worddsk}; \qquad t_{mem} = \frac{s_{blk}}{4} \times t_{wordmem}$$

Consider the following configuration.

Configuration 1

T = 1 Gbytes; M = 64 Mbytes; $N_b = 48$; $s_{blk} = \frac{64M}{3}$

$s_{pkt} = 48$; $t_{pkt} = 10\mu s$; $bw_{dsk} = 5$ MBps (SCSI-2)

For these physical parameters, we have that $t_{dsk} = 4,26s$ and $t_{net} = 4,44s$. Thus, the time to retrieve a text block from disk (sequentially) is identical to the time to retrieve a text block across the network. Further, this is also true if (a) $s_{pkt} = 96$ and $bw_{dsk} = 10MBps$ (fast SCSI-2) or (b) $s_{pkt} = 192$ and $bw_{dsk} = 20MBps$ (fast-wide SCSI-2). Since network bandwidth is increasing faster than disk transfer rate, we should expect that $t_{net} < t_{dsk}$ in the near future. For our analysis in this paper, we assume that $t_{net} \approx t_{dsk}$.

Let,

n_{sis}: number of sistrings in a text block or pat array ($n_{sis} = \frac{s_{blk}}{4}$)
t_{qck}: average time to generate (in memory) a pat array for a text block
$t_{bsearch}$: time to search (in memory) for the position of a sistring in a pat array

The parameter $t_{bsearch}$ accounts for the time to determine the position in a pat array of a given sistring. The pat array and its corresponding block of text are assumed to be resident in the local memory. This binary search reveals the position in the pat array for inserting a pointer to the given sistring (such that the pat property of lexicographical order is not violated). In the best case, this search requires a single comparison of sistrings. In the worst and average cases, $\log_2 n_{sis}$ comparisons are required. To simplify our analysis, we assume that comparing two sistrings requires comparing simply their first 4 characters (i.e., the first four bytes in the sistring)[3]. Given these considerations, each comparison requires retrieving 8 bytes from memory (4 for the pat array pointer and 4 for the first four characters of the sistring). Thus,

$$t_{bsearch} = \frac{(4+4)\log_2 n_{sis}}{bw_{mem}}$$

Consider configuration 1 above. Typical memory bandwidth nowadays is 200 MBps (megabytes per second) [8]. In this context, $t_{bsearch} = 900ns$ (nano seconds) which is larger than the time $t_{worddsk}$ to retrieve a 4 bytes word from disk ($800ns$ for SCSI-2 at typical transfer rate) or from a remote memory across the network ($833ns$). This shows that $t_{bsearch}$ is significative and cannot be ignored (as one might be inclined to do by considering that the binary search involves solely local memory accesses).

Computation of an entire array of counters requires executing the above binary search n_{sis} times. Let,

t_{bin}: time to perform n_{sis} binary searches in a pat array

Since $n_{sis} = \frac{s_{blk}}{4}$,

$$t_{bin} = t_{bsearch} \times \frac{s_{blk}}{4}$$

[3] In a random text with an alphabet of size $2 * 26 + 10$, the probability that a comparison of two sistrings requires comparing more than 4 characters is $(\frac{1}{62})^4$.

3.3.2 Execution Time for the *Patseq* Algorithm

The total execution time t_{Patseq} for the *Patseq* algorithm is composed of the following times:

> t_{pat}: time to retrieve the text blocks from disk, generate the pat arrays for each of them, and store these pats on disk
> $t_{counter\_computing}$: time to generate the arrays of counters and store them on disk
> $t_{pat\_merging}$: time to merge the pat arrays into the full (and final) pat

To generate the pat array for one text block it is necessary to (a) read the block from disk into memory, (b) determine the sistring pointers which compose the pat (this requires accessing the sistrings in memory and storing the pointers back to memory), (c) perform an indirect quicksort in the pat to sort the pointers, and (d) store the generated pat array on disk. Since there are N_b text blocks,

$$t_{pat} = (t_{dsk} + (2 * t_{mem}) + t_{qck} + t_{dsk}) * N_b$$

After the pat arrays are stored on disk, the counter_computing task initiates. This task ends with the computation of the composed counters. To compute the composed counter $C_{1+2+3+...+(j-1),j}$, it is necessary to: (a) read the pat p_j from disk (t_{dsk}), (b) read the block of text b_j from disk (t_{dsk}), (c) read $j-1$ blocks of text from disk (from b_1 to b_{j-1}) and for each of them execute n_{sis} binary searches (which allow incrementing the right counters), and (d) store the composed counter on disk (t_{dsk}). Since $N_b - 1$ composed counters have to be computed,

$$\begin{aligned} t_{counter\_computing} &= \sum_{j=1}^{N_b-1} [2 * t_{dsk} + ((j-1) * (t_{dsk} + t_{bin})) + t_{dsk}] \\ &= \frac{N_b^2}{2} * \frac{s_{blk}}{4} * t_{worddsk} + \frac{N_b^2}{2} * \frac{s_{blk}}{4} * t_{bsearch} + O(N_b) \end{aligned}$$

Following the counter_computing task, the pat_merging task is initiated. For generating the intermediary pat array $p_{1+...+j}$ it is necessary to: (a) read the pat array p_j from disk (t_{dsk}), (b) read the counter array $C_{1+...+(j-1),j}$ from disk (t_{dsk}), (c) read the pat array $p_{1+...+(j-1)}$ (which has been previously computed) sequentially from disk merging it with p_j on the fly, and (d) store on disk the pat array $p_{1+...+j}$. This whole process must be repeated $N_b - 1$ times. Remark that the original text block is not necessary anymore. Therefore,

$$\begin{aligned} t_{pat\_merging} &= \sum_{j=1}^{N_b-1} [t_{dsk} + t_{dsk} + (j-1) * t_{dsk} + j * t_{dsk}] \\ &= N_b^2 * \frac{s_{blk}}{4} * t_{worddsk} + O(1) \end{aligned}$$

Therefore, the total execution time for *Patseq* is given by

$$t_{Patseq} = \frac{3 * N_b^2}{2} * \frac{s_{blk}}{4} * t_{worddsk} + \frac{N_b^2}{2} * \frac{s_{blk}}{4} * t_{bsearch} + O(N_b)$$

3.3.3 Execution Time for the *Patseqpar* Algorithm

The execution time $t_{Patseqpar}$ for the *Patseqpar* algorithm is also composed of the times t_{pat}, $t_{counter\_computing}$, and $t_{pat\_merging}$ with the following differences: (a) t_{pat} is cut by a factor of N_b (because all pats are generated in parallel), and (b) except for the first step of the counter_computing task (which computes $C_{1,2}$) and the last step of the pat_merging task (which computes $p_{1+\ldots+N_b}$), these two tasks are run in parallel (see Figure 1). Thus, the execution time for the *Patseqpar* algorithm is composed of

$t_{parallel\_pats}$: time to compute the pats in parallel (no need to store them back to disk)
$t_{counter\_1\_pat\_N}$: time to compute $C_{1,2}$ plus time to compute the pat $p_{1+\ldots+N_b}$
$t_{counter\_pat\_par}$: time to compute the remaining portions of the counter_computing (no need to store counters back to disk) and the pat_merging tasks in parallel

These times are given by

$$\begin{aligned}
t_{parallel_pats} &= t_{dsk} + 2 * t_{mem} + t_{qck} \\
t_{counter_1_pat_N} &= (t_{dsk} + t_{bin}) + (2 * (N_b - 1) * t_{net}) \\
t_{counter_pat_par} &= \max\left((N_b - 2) * (t_{dsk} + t_{bin}), \sum_{j=2}^{N_b-1} (2 * (j-1) * t_{net})\right) \\
&= N_b^2 * \frac{s_{blk}}{4} * \frac{4 * t_{pkt}}{s_{pkt}} + O(N_b)
\end{aligned}$$

The expression for $t_{counter\_pat\_par}$ requires clarification. The counters are computed in parallel and thus, their computation time is given by the time to compute the more involving counter which is $C_{1+\ldots+(N_b-1),N_b}$. Since C_{1,N_b} had already been computed (in parallel with $C_{1,2}$), only $N_b - 2$ text blocks need to be retrieved. This is the first expression inside the *max* function. The second expression inside the *max* function is relative to the merging of the pats. Such operation has to be done in strictly sequential fashion. First compute p_{1+2}, second compute p_{1+2+3}, and so on. Computation of p_{1+2} requires retrieving p_1 from machine 1 (across the network) and shipping half of p_{1+2} back to machine 1 (also across the network). Thus, 2 text block transfers across the network are required. Analogously, the computation of p_{1+2+3} requires 2*2 block transfers. In general, computation of $p_{1+\ldots+j}$ requires $2 * (j - 1)$ block transfers.

The total execution time is given by

$$t_{Patseqpar} = N_b^2 * \frac{s_{blk}}{4} * \frac{4 * t_{pkt}}{s_{pkt}} + O(N_b)$$

4 *Patpar*: A Distributed Parallel Mergesort for Generating Pat Arrays

In this section we present a distributed parallel mergesort algorithm for the generation of pat arrays which is referred to as *Patpar*. The algorithm uses the aggregate memory of a cluster of workstations to store the full text and its pat array and thus, does not require accesses to local disks. We demonstrate that it improves the expected execution time by a factor of N_b.

4.1 A New Algorithm – *Patpar*

The *Patpar* algorithm works as follows. The text is partitioned in N_b blocks of size s_{blk}. Each text block b_i is assigned to a distinct workstation such that the corresponding pat array p_i can be generated locally (using, for instance, quicksort). After the generation of these pat arrays, the *ith* machine contains the block of text b_i and its pat array p_i in its main memory. Our idea is to merge these pat arrays by moving sistrings solely in the aggregate memory (i.e., without ever storing them on disk). Since each pat array is already sorted, we adopt the *mergesort* algorithm to perform this distributed merging.

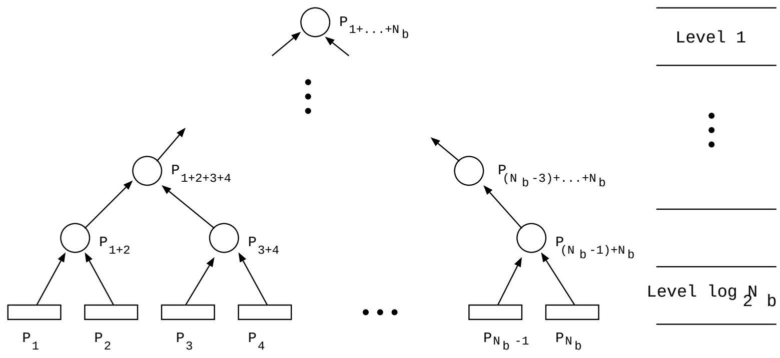

Figure 2: Execution strategy for the distributed parallel mergesort algorithm.

Figure 2 illustrates the merging procedure. At the bottom level (i.e., level $\log_2 N_b$), we group the machines in pairs. The lowest numbered machine in each pair controls the merging operation. The resultant pat array is stored in the aggregate memory of this pair of machines. The total number of merges is $\frac{N_b}{2}$ and they can all be done in parallel. We assume that the pat array pointers are words of 32 bits which already include absolute text offsets such that they do not need to be adjusted when they are moved from one machine to another. The algorithm for merging a pair of neighbor pat arrays p_{2t-1} and p_{2t} is as follows.

Algorithm 5 *Merging of the pat arrays p_{2t-1} and p_{2t} by machine $2t-1$*

$i = 2t - 1;\ j = 2t;\ s = 0;\ sj = 0;$
for $(si = 0;\ si \leq n_i;\ si++)$ {
while $(sistring(p_j[sj]) <= sistring(p_i[si]))\ p_{i+j}[s++] = p_j[sj++];$
$p_{i+j}[s++] = p_i[si];$
};

We assume the presence of proper sentinels at the end of the pat arrays p_i and p_j which make the test for array limits unnecessary. Further, the code above does not make it explicit that the resultant pat array p_{i+j} is stored in the aggregate memory formed by the *ith* and *jth* machines. We deal with such details in Subsection 4.2 where the execution time of the above algorithm is analyzed.

At the immediately above level (i.e., level $(\log N_b) - 1$), we group two pairs of machines into quadruples. At the next above level, eight machines are grouped together, and so on. At the root node, the final merge is done and the pat array $p_{1+2+...+N_b}$ generated.

4.2 Modeling the Execution Time

The execution time t_{Patpar} of the *Patpar* algorithm is composed of the following two times: (a) time to compute the pat arrays $p_1, \ldots, p_{N_b}$ in parallel and (b) time to merge the pat arrays in the aggregate memory. The first of these times is the $t_{parallel\_parts}$ time computed in Subsection 3.3.3. The second of these times depends heavily on the number of message packets exchanged which means that care must be exercised.

We first observe that as the computation moves up in the execution tree (Figure 2), a given pat array pointer and its respective sistring might end up residing in the memories of two distinct machines (because pointers are moved in the aggregate memory but sistrings are not). Given this observation, consider first a naïve approach for merging the pat arrays. In this naïve approach, the processor in control of the merging operation retrieves a remote pointer and its corresponding remote sistring according to the following protocol:

(1) send out a packet requesting the pointer
(2) receive a packet with the requested pointer
(3) send out a second packet requesting the corresponding sistring
(4) receive a packet with the requested sistring

In this scenario, the retrieval of a single remote pair $[pointer; sistring]$ requires an exchange of 4 packets. Since packets are 48 bytes long (current ATM technology), too much network bandwidth is wasted. A better approach is possible.

The weakness of the above naïve approach is that it fails to observe that: (a) the remote pat array is already sorted which allows packing at least 12 pointers in a single packet ($\frac{48}{4}$) and (b) it is not necessary to retrieve the whole remote sistring at once (because its first 4 characters might be enough for deciding a comparison). Our alternative approach takes advantage of these observations and works as follows.

We divide the available memory in 4 portions (instead of 3 as before) and reserve one of them for text. Thus, the number N_b of text blocks is now given by $\frac{4T}{M}$ which implies that our parallel algorithm requires more machines than the *Patseqpar* algorithm. To make this distinction clear we refer to the number of text blocks used by our algorithm as $\mathcal{N}_b$. Clearly, $\mathcal{N}_b = \frac{4N_b}{3}$. Also, the block size changes. Let $\mathcal{S}_{blk}$ be the block size used by our algorithm. Then, $\mathcal{S}_{blk} = \frac{3s_{blk}}{4}$. The 4 portions of memory are used to hold: (1) a block-size portion of the resultant pat array which is currently been computed, (2) the pat array which is participating in the current merging operation, (3) a block of text b_i, and (4) its respective pat array p_i. The pat array p_i is kept in memory throughout the computation to allow accessing the sistrings in b_i in lexicographical order. Remote pat array pointers and sistrings are now retrieved according to the following protocol:

(1) send out a packet requesting $\frac{48}{4}$ adjacent pointers
(2) receive a packet with the 12 pointers requested
(3) for each of the received pointers do:
(3a) send out a packet (to the proper machine) requesting $\frac{48}{4}$ sistrings of size 4 sorted lexicographically
(3b) receive a packet with the requested sistrings

A minimum extra buffer space (of size $48 * \mathcal{N}_b$) is required to hold the newly arrived data (i.e., a buffer of size 48 bytes is reserved locally for each remote machine in the network). As a result of the above protocol, the next 12 remote pointers (and their respective sistrings) are promptly available for the merging operation. Furthermore, since the sistrings in each machine buffer are sorted lexicographically, they can be processed sequentially as the merging operation moves on.

A problem arises when the first 4 characters of a sistring are not enough to decide a comparison. In this case, an extra request message is dispatched to the proper machine requesting the whole sistring (which fits in a single packet). This event happens with a probability q which is usually small ($(\frac{1}{62})^4$ in random texts with an alphabet size of 62) and thus, the number of extra messages for retrieving full sistrings is expected to be small.

The time $t_{parallel\_merging}$ to conclude the whole merging task can be computed as

$$\begin{aligned} t_{parallel\_merging} &= \sum_{i=1}^{\log \mathcal{N}_b} \left[2 * ((2^i - 1) * t_{net} + (2^i - 1) * t_{net}) + (2^i - 1) * t_{net}\right] \\ &= 10 * \mathcal{N}_b * \frac{\mathcal{S}_{blk}}{4} * \frac{4 * t_{pkt}}{s_{pkt}} + O(\log \mathcal{N}_b) \end{aligned}$$

Since $\mathcal{N}_b = \frac{4N_b}{3}$ and $\mathcal{S}_{blk} = \frac{3s_{blk}}{4}$, the total execution time for our parallel algorithm is given by

$$t_{Patpar} = 10 * N_b * \frac{s_{blk}}{4} * \frac{4 * t_{pkt}}{s_{pkt}} + O(\log N_b)$$

It can also be shown that the extra time t_{extra} spent retrieving full sistrings is given by

$$t_{extra} = \mathcal{N}_b * \frac{q * s_{pckt}}{2} * t_{net}$$

which can be disregarded in the calculation of the total execution time if q is not excessively high.

5 Analytical Insights

In this section we briefly compare the execution times of the three algorithms studied.

We already know that

$$t_{worddsk} \approx t_{bsearch} \approx \frac{4t_{pkt}}{s_{pkt}}$$

and that these times should remain comparable even with improvements in disk and memory bandwidth because those can be compensated by improvements in the network bandwidth.

We first compare the execution times for the Patseq and Patseqpar algorithms. Examination of the execution time equations leads to

$$t_{Patseq} \approx 2 * t_{Patseqpar}$$

This improvement in execution time is due to the fact that the *Patseqpar* algorithm is able to run the counter_computing task in linear time. The improvement is not considerable because of the inherently sequential nature of the *Patseq* algorithm.

We now compare the execution times for the three algorithms. The execution time equations allow us to write

$$t_{Patpar} \approx \frac{10}{N_b} * t_{Patseqpar} \approx \frac{10}{2 * N_b} * t_{Patseq}$$

Consider first the situation in which the number of machines needed for computing the pat array for the whole text is small (~ 10). In this trivial case (in which the size of the available memory is of the same order of magnitude of the text size), the three algorithms present execution times which are comparable and the *Patseq* algorithm is preferrable because its implementation is simpler.

When the text size far exceeds the size of the memory available (the most common situation for large texts), the factor N_b increases considerably and Patpar becomes the algorithm of choice. For configuration 1, for instance,

$$t_{Patpar} \approx \frac{t_{Patseqpar}}{4.8} \approx \frac{t_{Patseq}}{9.6}$$

which is an almost tenfold improvement over the sequential algorithm. As the number of machines increase (due to the low availability of machines with large memories[4]) the improvements provided by Patpar become more accentuated. For instance, if the machines in configuration 1 are downsized to 16 Mbytes in available memory ($N_b = 192$) then

$$t_{Patpar} \approx \frac{t_{Patseqpar}}{19.2} \approx \frac{t_{Patseq}}{38.4}$$

which is an improvement of more than one order of magnitude over the Patseqpar algorithm. Furthermore, increases in the text size also favor Patpar. For instance, consider that $T = 4Gbytes$ in configuration 1. Then, again

$$t_{Patpar} \approx \frac{t_{Patseqpar}}{19.2} \approx \frac{t_{Patseq}}{38.4}$$

Finally, we also point out that the Patpar algorithm does not have the stringent disk space requirements of the Patseqpar algorithm. It does uses more machines than Patseqpar, but machine availability is always high [1] while disk availability is not.

6 Conclusions

We proposed a new algorithm for the parallel generation of large pat arrays in a high-bandwidth network of workstations. Further, we discussed a well known sequential algorithm (which is very efficient running on a single machine) and a parallel implementation for it. We carefully analyzed the complexity of the execution times for the three algorithms. Considering that the exchange of a small message at the application level can be done in about $10\mu s$ (which is still not a reality but which is coming), we showed that our algorithm is clearly preferrable to the other two (when the algorithms run in commercial workstations available nowadays) and that the improvements in execution time it provides are expected to go up to one order of magnitude and beyond.

References

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

[4] Memory is nowadays the most expensive component of a computer. Doubling the memory of a Sun Sparc4 workstation from 32M to 64M increases the cost of the machine in 50%.

[2] 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.

[3] 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.

[4] 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.

[5] Donald E. Knuth. *The Art of Computer Programming: Sorting and Searching.* Addison Wesley, 1973.

[6] Jeffrey Kuskin et al. The stanford FLASH multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 302–313, Chicago, IL, April 1994.

[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] Sun Microsystems. Sun WWW home page, March 1996. http://www.sun.com/.

[9] D.R. Morrison. PATRICIA – Practical Algorithm To Retrieve Information Coded In Alphanumeric. *JACM*, 15(4):514–534, October 1968.

[10] Robert Sedgewick. *Algorithms.* Addison Wesley, second edition, 1988.