Network and Memory Analysis in Distributed Parallel Generation of Pat Arrays

João Paulo W. Kitajima
Berthier Ribeiro
Nivio Ziviani

Departamento de Ciência da Computação
Universidade Federal de Minas Gerais
e-mail: {kitajima,berthier,nivio}@dcc.ufmg.br

Abstract

The performance of parallel and distributed algorithms for generation of large pat arrays is analyzed. These algorithms are evaluated taking into account a high-bandwidth network of workstations, a TCP/IP-based network and an heterogeneous network with different memory sizes. In the first case, performance of the parallel versions are significantly better. In the second case, the sequential algorithm is clearly the best. In the third case, different memory sizes will hardly improve execution times significantly.

1 Introduction

The traditional model of text in information retrieval (IR) is that of a collection of documents indexed by keywords. In this model, the user specifies his information need by providing sets of keywords and the information system retrieves the documents which best approximate the user query. Further, the information system might attempt to rank the retrieved documents using some measure of similarity. 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 IR. 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. 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. 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, 4]. 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. In this paper we investigate (a) the impact of the TCP/IP protocol, and (b) the impact of heterogeneity on the performance of the parallel versions of an algorithm for pat arrays generation presented in [3]. "Heterogeneity", in our context, means machines with different memory sizes. Memory availability is one of the main concerns when developing applications that build indices for large texts (greater than 1 gigabyte).

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 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.

3 The *Patseq* and *Patseqpar* Algorithms

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*.

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., $N_b = \lceil \frac{T}{B} \rceil$). 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. The merging of the N_b pat arrays is then 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 1 <u>*Naïve algorithm for merging partial pat arrays.*</u>

$P = p_1$; /* 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 2 <u>*Computing the array of counters $C_{i,j}$.*</u>

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 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

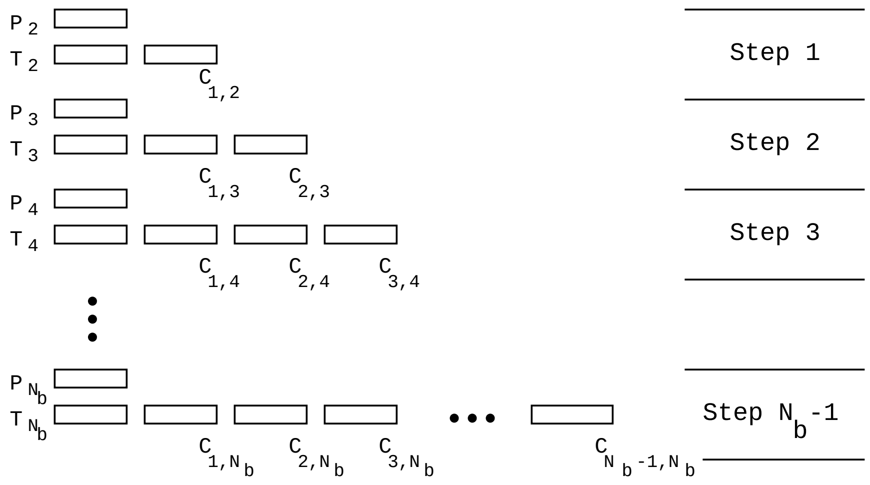

Figure 1: Data structures used by the *Patseq* algorithm for merging the N_b pat arrays.

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 3 *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* can not be improved. However, the expected execution time can be improved by parallelizing the counter_computing task.

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 2 (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

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

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). 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 the configuration above. Typical memory bandwidth nowadays is 200 MBps (megabytes per second) [5]. 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 can not 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 $n_{sis} = \frac{s_{blk}}{4}$,

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

3.3.2 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

The total execution time for *Patseq* is given by

$$t_{Patseq} \approx \frac{3 * N_b^2}{2} * \frac{s_{blk}}{4} * t_{worddsk} + \frac{N_b^2}{2} * \frac{s_{blk}}{4} * t_{bsearch}$$

3.3.3 Execution Time for the *Patseqpar* Algorithm

The execution time $t_{Patseqpar}$ for the *Patseqpar* algorithm is 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

$$\begin{aligned} t_{parallel\_pats} &= t_{dsk} + 2 * t_{mem} + t_{qck} \\ t_{counter\_1\_pat\_N} &= (t_{dsk} + t_{bin}) + (2 * (N_b - 1) * t_{net}) \\ t_{counter\_pat\_par} &= \max\left((N_b - 2) * (t_{dsk} + t_{bin}), \sum_{j=2}^{N_b-1} (2 * (j-1) * t_{net})\right) \\ &\approx N_b^2 * \frac{s_{blk}}{4} * \frac{4 * t_{pkt}}{s_{pkt}} \end{aligned}$$

The total execution time is given by

$$t_{Patseqpar} \approx N_b^2 * \frac{s_{blk}}{4} * \frac{4 * t_{pkt}}{s_{pkt}}$$

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

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 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. 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 $p_{1+2+\ldots+N_b}$ generated.

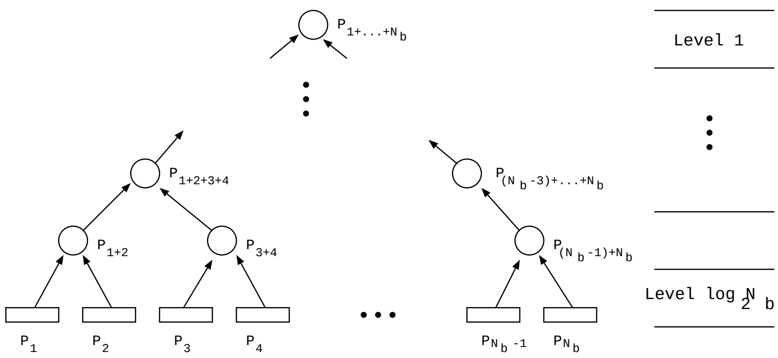

Figure 2: Execution strategy for the distributed parallel mergesort algorithm.

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 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

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

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

$$t_{Patpar} \approx 10 * N_b * \frac{s_{blk}}{4} * \frac{4 * t_{pkt}}{s_{pkt}}$$

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 Conclusion: Network and Memory Considerations

5.1 TCP/IP Networks

The high-bandwidth analysis considers user-to-user communication latencies in the order of 10 μs. Unfortunately, common Ethernet networks and even specialized switched networks of machines like IBM SP systems do not provide latencies with this order of magnitude. Experiences on the old IBM SP-1 and the more recent IBM SP-2 (or just SP) systems give us times latencies varying from 2 miliseconds to 10 miliseconds (user-to-user latencies in the worst case) [2, 6]. This means two to three orders of magnitude larger than latencies in a high-bandwidth network. Naturally, the gains of one order of magnitude in high performance networks are lost when working with conventional technology: *Patseq* (the original sequential algorithm) is the best algorithm under these conditions.

5.2 Influence of Memory Size

It is quite difficult to work in a totally homogeneous network of workstations, with the same processor, same memory sizes and even same operating system. In our analysis, CPU heterogeneity is not a problem because the pat construction is an application typically I/O-bound. We remark that the quantitative analysis given by this paper ignores the processing overhead. Disks unities can also vary from workstation to workstation but, due to standard interfaces (e.g, SCSI), disk bandwidth is reasonably homogeneous in a network. The exception here is the case where file systems are mounted through the network. If network goes faster than disk, the effective bandwidth is still given by the disk unit. In the contrary, network is the bottleneck. However contention exists and global bandwidth is surely worse.

On the other hand, memory may vary from machine to machine. $t_{Patseqpar}$ and t_{Patpar} are insensitive to memory access times when the number of blocks is high (e.g., superior to 10). The parameter of importance is t_{net}. So, the only way memory can affect the performance when exploring high parallel systems is when memory size is different from machine to machine. If the proposed parallel algorithms work without changes, machine with larger memories will suffer of subutilization (in the case memory is totally given to the pat construction algorithm and operating system).

In *Patseqpar*, larger memories can eventually support larger pieces of the already sorted partial pat. In:

$$t_{counter\_pat\_par} = \max\left((N_b - 2) * (t_{dsk} + t_{bin}), \sum_{j=2}^{N_b-1}(2 * (j-1) * t_{net})\right)$$

the first argument of *max* is not affected by a larger memory: disk is still read sequentially and binary searches are done also sequentially. One possibility is to perform in parallel, in the same machine, binary search and disk access. t_{dsk} may, in this case, be transparent to the machine computing $C_{1+2+...+(j-1),j}$. Some gain could be also obtained if partial sorted pats do not go back to their original machines. They rest in the machine where merge has been done. But gain is useless: dropping the second factor by half, for example, means that first processor has no additional memory, second processor can support three original text blocks plus one, third processor can support three original text blocks plus two, and so on. Fourth processor and next processors will have at least double memory than the smaller machine. In this case, half of the processors can be used because this half will surely have twice primary memory than the original configuration. In the above cases, complexity is not changed: the algorithm remains *quadratic* in the number of blocks.

In *Patpar*, the same constraint is valid. If half processors has primary memory twice larger than the other machines, it is better to use only these "larger" machines. If only some machines are larger than others, these machines have to be the "master" processors when merging pats. These machines can simulate virtual processors putting text and pats in the additional memory. However, gain is limited due to the synchronization needed between levels $1, 2, \ldots, \log_2 \mathcal{N}_b$. If in *Patseqpar* we have an application typically sequential, in *Patpar* however, we have levels that are sequential. The additional memory will be source of load imbalance with processors waiting for other processors. Finally, some "master" processors do not remain "master" in the next level. In the above scheme, additional memory of these ex-"master" processors is useless.

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] Laurent Azema. Evaluation de stratégies d'ordonnancement statique sur ordinateurs à mémoire distribuée, June 1995. DEA thesis - Institut National Polytechnique de Grenoble - Grenoble, France.

[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] 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.

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

[6] Ronald Mraz. Reducing the variance of point-to-point transfers for parallel real-time programs. *IEEE Parallel and Distributed Technology*, 2(4):20–31, 1994.