Theory Comput. Systems **32**, 591–623 (1999)

Theory of Computing Systems

Optimal High-Performance Parallel Text Retrieval via Fat-Trees*

B. Mamalis,[1] P. Spirakis,[1,2] and B. Tampakas[1,3]

[1] Computer Technology Institute,
Kolokotroni 3, 26221 Patras, Greece
{mamalis,spirakis,tampakas}@cti.gr

[2] Computer Engineering and Informatics Department,
Patras University,
26500 Rion, Patras, Greece

[3] Technological Educational Institute (TEI) of Patras,
M. Alexandrou 1, Koukouli, 26334 Patras, Greece

Abstract. We present here a high-performance parallel free-text retrieval method for multiple text queries using the Vector Space Model. Our method employs the fat-tree area universal routing network for connecting the processors of a parallel machine, however (in its general form) it could also be efficiently applied over any other high-bandwidth network of processors or workstations. We provide a theoretical analysis of our technique which shows it is excessively efficient and clearly superior (concerning both the amortized processing times and the average waiting times per query) to parallel text retrieval methods for single queries (e.g., based on binary trees). Moreover, we prove our method to be optimal with respect to the execution of all the implied communication tasks on ideal fat-trees. We also experimentally demonstrate the high performance and superiority of our technique via suitable embeddings of ideal fat-trees on realistic two-dimensional mesh-oriented parallel machines (e.g., the GCel Parsytec machine) and via the use of the large-scale TREC document collections. Note that the fat-tree can simulate any other network of the same hardware with only a polylogarithmic loss of efficiency.

* This research was partially supported by ESPRIT LTR Project No. 20244 (ALgorithms & COMputation in Information Technology—ALCOM-IT) and by No. 9072 Basic Research Project (GEPPCOM) of the European Union (EU).

1. Introduction

The incorporation of parallelism in large-scale software applications is usually regarded as an effective solution toward the direction of gaining fast and interactive response times. Naturally, many attempts have been made over the last decade to apply parallel processing techniques on large-scale document retrieval systems and text database searching (see Section 2). Most of these attempts are mainly aimed at building efficient multiprocessor text retrieval systems and try to meet the following basic requirements:

1. Low total communication times.
2. High retrieval effectiveness (in terms of recall and precision).
3. Interactive response times within a friendly user interface environment.
4. Significant response-time improvements compared with serial machines.

In our work, in order to satisfy the above requirements, we have developed a high-performance parallel text retrieval algorithm for multiple text queries using a fat-tree as the underlying interconnection network (following the area universal model described in [14] where only the leaves serve as working processors). Generally, fat-tree structures guarantee the efficient exchanging of multiple messages due to their high-capacity communication channels. Trying to benefit from this basic fat-tree feature, we have followed the approach of the concurrent processing of multiple queries together, assuming that we have an interactive text retrieval system under a heavy query load (e.g., a highly accessed search engine over the Internet). In this case, the total number of messages that have to be exchanged increases in proportion to the number of queries. Specifically, for N queries and N working processors, a total of $O(N^2)$ different messages have to be concurrently exchanged among all the processors. However, it is possible, via the specific optimal communication protocol that we use, to achieve the delivery of all the messages in very low total communication times via an ideal fat-tree-structured network. Similar results could also be obtained through the use of several other high-capacity networks of processors or workstations (the fat-tree topology has been chosen as the most challenging and suitable paradigm) making our method quite general and useful for extended multiple queries text retrieval applications.

As a direct result, the amortized communication time per query is proved (both theoretically and experimentally) to be excessively small and almost constant ($O(1 + \log N/N)$) with regard to the number of processors. Moreover, the total amortized query processing times are proved to be much better than those offered by the usual single-query algorithms, whereas the average query waiting times are also substantially improved. Furthermore, the specific communication protocol used for multiple message exchanging is proved to be optimal (in the sense of exact analysis) for a general class of ideal fat-trees. Finally, trying to obtain some experimental evidence for the high performance of our fat-tree algorithm on a realistic parallel environment, we have performed experiments over the GCel Parsytec machine [28] with the use of a considerably efficient embedding (described in the Appendix) of the ideal fat-tree topology on a two-dimensional mesh-structured network. The above experiments have led to quite satisfactory experimental results, that are clearly better than the corresponding ones of the previously used binary-tree single-query algorithm stated in [7] and [17].

In the next section the most recent related work is briefly presented together with a short description of the well-known fat-tree network interconnection model. The motivation of the problem, the detailed presentation of our multiple-query processing algorithm for fat-trees and the performance analysis of the algorithm then follow (Section 3). In Section 4 we theoretically show the superiority of the multiple-queries algorithm over the usual single-query algorithms for binary trees, concerning both the amortized processing times and the average waiting times per query. The optimality of the communication tasks of the algorithm as well as an extension of the algorithm appropriate for a general class of ideal fat-trees are stated in Section 5. Finally, in Section 6 we present the experimental results of our work, as well as a short comparison with the results offered by the single-query algorithm presented in [7] and [17].

2. Background and Related Work

Many valuable attempts have been made with regard to parallel text retrieval systems during the last decade. These attempts vary concerning the text processing model they use (signatures, inverted file structures, vector space model) and the type of parallel environment they adopt.

With regard to the text processing model, initial parallel implementations were based on overlap-encoded signatures [5], [18], [19], [24]. Other attempts have been made using frame-sliced partitioning on signature files [10]. However, the various limitations of these methods (i.e., constrained interactive access for large signature files [26], only binary document weights supported [19]) led to studies and implementations of parallel inverted file structures [23], [25]. These attempts using the inverted file model [29] showed considerable performance, even under the above limitations and the limited power parallel environment they used. Concerning the type of parallelism, we can distinguish between the SIMD and MIMD environments (due to Flynn [8]). Most parallel text retrieval systems [23]–[25] have been implemented on the earlier versions of the Connection Machine (CM-1 and CM-2), which are classical types of SIMD architectures using pipelined vector processors. However, the last few years have seen increasing interest for parallel retrieval systems under MIMD environments, mainly using microprocessor networks where each processor had its own local memory [5], [6], [22]. High-performance transputer networks with message passing communication mechanisms have been used and they have led to very efficient implementations [6]. The development of the Parallel InfoGuide system [1] has been regarded as a quite valuable attempt in the field, mainly aiming at building an efficient distributed-memory multiprocessor system appropriate for gaining very fast query response times.

Finally, the PFIRE system (presented in our previous work [7], [17]) is a parallel information retrieval system running on the Parsytec GCel3/512 supercomputer, and it can be regarded as the parallel extension of the previously existing conventional FIRE text retrieval system (Full Information Retrieval Engine [12], [27]). PFIRE uses a very efficient binary-tree parallel text retrieval algorithm based on the Vector Space Model (VSM [21]), and provides up to 90% utilization when all the 512 transputers are used with the primary storage full of data. In our work PFIRE mainly serves as the basis for

the comparative evaluation of our fat-tree multiple-queries algorithm to the binary-tree single-query algorithm of [7] and [17].

2.1. *Fat-Tree Structures*

Fat-trees [3], [14] consist of a general class of the area universal networks which, as is already proved [14], can simulate any other network of the same amount of hardware spending slightly more time (only by a polylogarithmic factor greater). Moreover, a fat-tree of a given size (due to its universality properties) can be regarded as the best routing network of that size. Fat-tree structures have become quite popular lately (e.g., the CM-5 machine uses a fat-tree as its interconnection pattern [15]). In its general form (see Figure 1), a fat-tree of size N is a binary tree (of depth $h = \log N$) with N leaves which are the working processors of the fat-tree. The $N - 1$ internal nodes are concentrator switches [14] which support efficiently the communication among the working processors using high-capacity channels that connect them to each other. The capacity of these bidirectional channels is progressively increased with the increase of the height of the tree. Also there is a host processor (following the model described in [14]) that connects the whole fat-tree to the outside world.

In our work we initially assume an ideal configuration [4] of a fat-tree where the capacity of the channels (see Figure 1) is doubled on each level of the tree. That means that each internal node on the kth level has capacity (number of parallel bidirectional links) equal to 2^k, and this capacity is shared equally to its two children. More formally, this specific class of ideal fat-trees can be described (similarly to [4]) by the general function: $[c(n) = n]$, which means that the capacity of each node (the sum of the capacities of its two channels) of the fat-tree (having n leaves in its own subtree—$c(n)$) is equal to the number of leaves of the corresponding subtree (n). Consequently, the root capacity is equal to the number of the leaves (N) of the whole fat-tree. In this work, we also deal with a more general class of ideal fat-trees with node-capacity function equal to $c(n) = n/2^i | i$ level of the node and $i = 0 \cdots \log N - 1$.

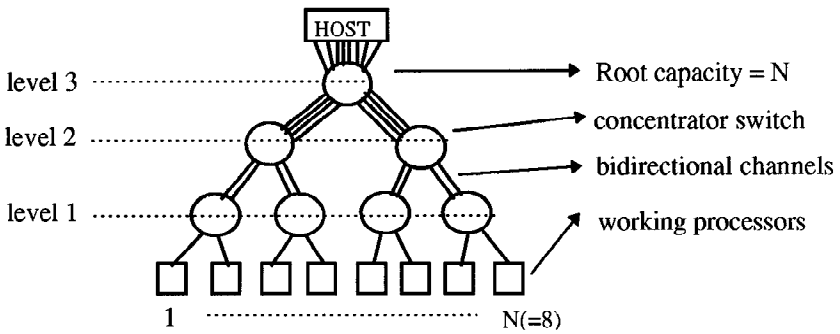


Fig. 1. An ideal fat-tree structured network of size $N = 8$.

3. Our Parallel Multiple Text Query Processing Approach

3.1. *The Parallel VSM-Based Text Retrieval Problem*

Generally, our work deals with the problem of finding an efficient solution for the parallelization of the text retrieval tasks implied by the use of the Vector Space Model (VSM), in a distributed memory parallel environment. Following this model (and after the application of suitable *stopping* and *stemming* procedures, see [21]), each document D_i in the whole document collection (of size D) is represented (off-line automatic indexing phase) by a corresponding document-term vector of the following form:

Definition 3.1 (from [20] and [21]). A *document-term vector* D_i representing a document d in a document collection of size D and m different terms (word stems extracted out of all the D documents) in total, is an ordered set of m weight values ranging from "0" to "1", each one corresponding to one term and representing the *significance* of this term for the specific document:

$$D_i = (wt_{i1}, wt_{i2}, \ldots, wt_{i(m-1)}, wt_{im}).$$

The specific weight values assigned to each term and for each document are mainly based on the frequency of occurrences of each term in both the specific document D_i and the whole document collection. They are normally computed via some statistical weighting function and they usually range between "0" and "1", where "0" corresponds to terms not appearing in the specific document.

During the on-line retrieval phase each user query Q_i (a set of words) is indexed in the same way, thus resulting to a query-term vector of almost the same type:

Definition 3.2. A *query-term vector* q_i representing a user query Q_i is also an ordered set of n values, each one corresponding to one term of the document collection (like a document-term vector), with the difference that each value is usually set either to "0" or to "1" depending on the presence of the corresponding term in the user query or not.

However, a query-term vector may also consist of weight values ranging between "0" and "1", in the case that the user assigns weights on the query terms by himself. Afterwards, the query-term vector is compared with all the document-term vectors (scoring task) of the indexed collection separately via the cosine-similarity function (which is based on the computation of the inner product of two vectors [21]) and a corresponding document list containing the query-similarity values of all the documents is extracted. Finally this relevant documents list is ranked (ranking task) and the top R documents (they are supposed to be the most relevant) are presented to the user. Now, considering a distributed memory parallel environment of N working processors, the whole set of the D document vectors is supposed to be appropriately distributed and almost equally bal-

anced over the local memories of the N processors. Consequently, supposing that we are

— **given** a set X of concurrently arriving text queries which are represented by $|X|$ corresponding VSM-indexed query-term vectors of the following form:

$$q_i = (q_{i1}, q_{i2}, \ldots, q_{i(m-1)}, q_{im}),$$

— **the goal** is to find an appropriate algorithm and a corresponding well-suited interconnection network (connecting the N working processors), for the efficient parallelization of the scoring and ranking tasks (as mentioned above) for all the $|X|$ queries over all the D documents of the given text collection.

3.2. *General Items*

Naturally, a typical approach to the above problem could be sequential processing (manipulating the queries one by one) with the use of an appropriate single-query parallel algorithm [1], [5], [7], [10], [17], [19], [24], [25], for each query, i.e., the single-query algorithm presented in [7], achieves very good parallelization using a simple binary network topology for connecting the working processors. In our work, trying to minimize further the total communication times needed, we have followed a different approach, based on concurrent processing of all the $|X|$ queries in sets of N (equal to the number of processors) queries each time. In this case the communication messages needed for processing N queries together (for spreading the queries, collecting and merging the results) are gathered in one execution leading to a (quite heavy) total communication task.

Consequently, the appropriate use of high-capacity networks that can deliver multiple messages concurrently in very low total communication times would be a proper solution. Toward the above direction most of the well-known high-capacity parallel interconnection networks (fat-trees, hypercubes, butterflies [13]) could be efficiently used. In our work we have chosen the use of the ideal fat-tree topology (which additionally has the advantage of providing a number of significant area universality properties, see [14]) as a paradigm of a well-studied high-capacity network which is able to perform efficiently the concurrent exchanging of multiple messages needed for the whole execution of our multiple text queries algorithm. Moreover, the high-capacity of the communication channels offered by the ideal fat-tree topology is a quite challenging feature, making the fat-tree perhaps the most suitable network for the execution of the concurrent communication tasks of our algorithm. However, we have to note here that our multiple-queries algorithm is quite general and it could operate efficiently over any other high-capacity network of processors or workstations that could deliver multiple messages concurrently in very low total communication times.

The resulting fat-tree multiple-queries algorithm (optimal for a general class of ideal fat-trees) is proved to have clearly better total communication performance than the iterative sequential execution of a single-query algorithm. Furthermore, it gives very good results for both the amortized processing time and the average waiting time per query in an ideal fat-tree structured network with capacity function $c(n) = n$.

The use of our algorithm is strongly recommended especially in the case of a heavy query load. This is the case when the number of the concurrently arriving queries is much greater ($|X| \gg N$, e.g., $|X| = a * N | a \gg 1$) than the number of nodes (N) of the fat-tree, i.e., this is a usual phenomenon for a frequently accessed search engine over the

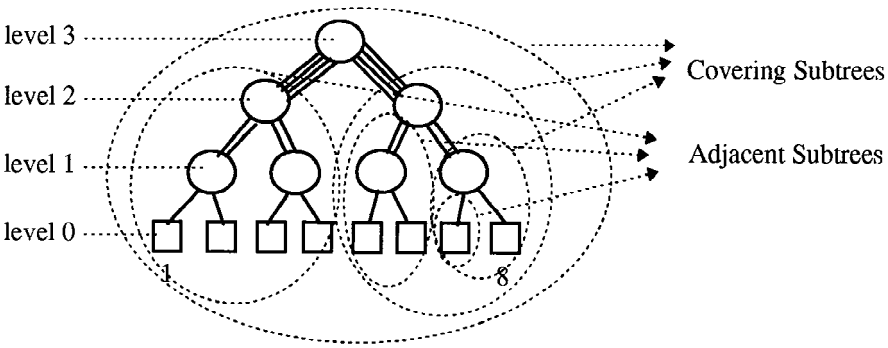


Fig. 2. Covering and adjacent fat-trees for processor 8.

Internet. In this case the fat-tree machine can complete the task by processing N queries each time and assigning (virtually) one query to each node. For the above reason we can work with N queries without restricting the generality.

3.3. *Presentation of the General Multiple-Queries Algorithm*

3.3.1. *Initial Settings and Assumptions.*

1. We are given a set of D document vectors indexed using the VSM text processing model.
2. We are given a set of N query vectors (indexed using VSM too).
3. The D document vectors are shared appropriately to all the N processors and the total amount of data is appropriately balanced. Without loss of generality we say that each processor finally holds approximately D/N vectors in his local memory.
4. There is an underlying ideal fat-tree network of size N and $c(n) = n$ (Figure 2) and one host node (with channel capacity equal to N) which holds the N query vectors. The processors of the fat-tree are numbered $1, \ldots, N$ from left to right.
5. Concerning the underlying fat-tree network, we define the notions of *covering* and *adjacent* subtrees (Figure 2) for each one of the N processors. The *covering subtrees* of one processor are the nonsimilar subtrees of increasing height that contain this processor. Actually there are $\log N$ covering subtrees for each processor of sizes (number of leaves) 2^k, $k = 1 \cdots \log N$. At level k of a covering subtree of node i, we call a subtree of the same parent (left or right positioned with respect to node i) that does not contain node i and contains 2^{k-1} leaves the *adjacent subtree* of node i at this level. For each node i there are in total $\log N$ such adjacent subtrees (of increasing height).

3.3.2. *The Parallel Algorithm.*

The algorithm is composed of the following rounds:

Round 1. The host processor sends the N query vectors to the N processors using appropriate pipelined steps. We assume that the leaves are named and this naming permits

the host to establish the relation between a query and the responsible leaf of the fat-tree. During each step the host processor sends the same query to all the leaves. That results in N messages for each step, which are delivered to the N leaves of the tree. At the end of all the steps each one of the N processors has received the complete set of N queries and it knows that he is responsible for a specific query. During round 1 we have in total N^2 messages sent to the leaves of the fat-tree (N messages to each leaf).

Round 2. Each one of the N processors performs the scoring and ranking tasks over his own D/N document vectors for all the N queries. That results in N ranked relevant document sets (e.g., of size R) for each query. Thus, after round 2, each processor has to collect the other $N-1$ relevant document sets that refer to the query he is responsible for (from the other $N-1$ processors), and proceed to the merging of the results.

Round 3. Each processor sends each of the $N-1$ (all the sets except the one that refers to its own query) relevant document sets (*RD-set* messages) to its corresponding, responsible for it, processor. This is performed via the following pipelined algorithm (it is executed concurrently by all processors):

for each node i do
for $k = 1$ *to* $\log N$ *do*

1. Send the appropriate relevant document sets to the $2^k - 1$ leaves of the kth adjacent subtree using the order specified in rules 2 and 3 below.
2. Send first to the adjacent neighbor of node i at level k. We specify as an adjacent neighbor at level k the node that has the same position as node i, at the adjacent subtree of level k (e.g., in Figure 3, the adjacent neighbor of node 2 at level 2 is node 4, and the adjacent neighbor of node 2 at level 1 is node 1).
3. For the following $2^{k-1} - 1$ steps, and in each step, send the appropriate relevant document sets to the rest of the leaves of the adjacent subtree of level k, using a left to right order and going circularly to the subtree when it necessary.

An example for $N = 8$ is given in Figure 3. During this round we have in total $N(N-1)$ messages exchanged between the leaves. At the end of round 3 each processor has received $N-1$ relevant document sets and it can proceed to the merging of the results for its own query.

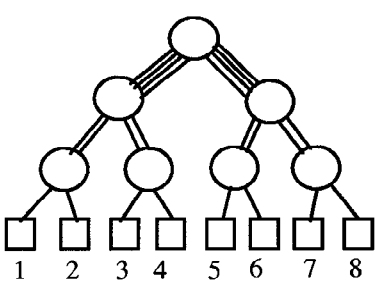


Fig. 3. An example of the execution of round 3 for $N = 8$.

Round 4. Each processor merges the $N-1$ relevant document sets that correspond to its query (the one it is responsible for), which have been received from the other processors during round 3. A final ranked relevant document set of size R is extracted from each responsible processor.

Round 5. Each processor sends its final relevant document set to the host processor. In this step we have N messages that are sent simultaneously from leaves to the root.

3.3.3. *The Performance Analysis of the Algorithm.* The first observation is that the computational tasks of scoring and ranking (round 2) are executed concurrently and for the whole set of queries. This simple algorithm property seems to guarantee the high level of speed-up. Since these tasks are both executed concurrently by all the nodes, the total required time is equal to the time for scoring and ranking N queries only in one processor and only over D/N document vectors. The parallelization of the computational tasks of round 2 is somewhat similar to the one presented in [7] if we consider each query separately. Specifically, assuming that T_s is the required time for scoring and T_{lr} for ranking on each processor for one query, in the single-query algorithm of [7] the time required for scoring and ranking is equal to $T_s + T_{lr} = 1/P \cdot (t_s + t_{lr})$ where P is the number of processors and t_s and t_{lr} are the serial times required for scoring and ranking, respectively (that means almost perfect parallelization). Consequently, the total computational time (scoring and ranking) required by our multiple-queries algorithm, where N queries are processed by each processor, can be directly computed by multiplying by N:

Proposition 3.1. *The computational time needed for the completion of scoring and ranking tasks of round* 2 *of our multiple-queries algorithm is equal to*

$$T_c = N \cdot (T_s + T_{lr}). \tag{1}$$

Regarding the merging of the results (round 4) we note that it succeeded in parallel and for all the queries. We achieve the maximum concurrency since all the processors are working in parallel and with equally weighted tasks. Therefore, the total time needed for merging is obviously equal to the merging time of one processor. Assuming that the size of each relevant document set is R and the required time for merging N ranked lists of size R is $O(N \cdot R)$, then the following statement holds:

Proposition 3.2. *The time needed for the completion of the merging task of round* 4 *of our multiple-queries algorithm, is equal to*

$$T_m = 2 \cdot (N-1) \cdot R. \tag{2}$$

Actually, in our case merging is performed progressively. The first list is merged with the second (time $2 \cdot R$), then we keep R docs in the result and we merge with the third and so on till the Nth list.

In the following we give the analysis of the communication times during rounds 1, 3, and 5. In these rounds there is an exchange of multiple messages, which, as we will see, does not significantly overhead the system thanks to the high capacity of the fat-tree channels.

Proposition 3.3. *The communication time T_{r1} needed for the execution of round* 1 *of our multiple-queries algorithm is equal to*

$$T_{r1} = N + \log N \quad \textit{query message steps}. \tag{3}$$

Proof. We note that if we had to send N query messages only to one processor, this would have been achieved (using pipelining) in $N - 1 + \log N$ steps, using only one path (of length $\log N$) from the root to this specific processor. However, as we can easily notice (e.g., from Figure 1, there are N disjoint paths (of length $\log N$) connecting the root with the leaves, one for each leaf $(1, \ldots, N)$. This is a basic property of the ideal fat-tree, based on the fact that the total capacity of each intermediate-level switch node is equal to the leaves of the subtree of this switch node. Thus the host can use these paths in parallel in order to send concurrently the N query messages to all the processors (one path for each processor) in $N + \log N - 1$ message steps. Thus if we add one step for the host-to-root communication we conclude to a total of $N + \log N$ steps. □

Proposition 3.4. *The communication time T_{r5} needed for the execution of round* 5 *of our multiple-queries algorithm is equal to*

$$T_{r5} = \log N + 1 \quad \textit{RD-set message steps}. \tag{4}$$

Proof. We note that during round 5 (a) only one message has to be sent from each leaf to the host and (b) the exchanged messages are of size R. Following an analysis method similar to the above (see the proof of Proposition 3.3), we conclude a total of $\log N + 1$ communication steps are needed for the whole execution. □

Considering now the communication time needed for the execution of round 3 of our algorithm, the reader may easily notice that we have N processors, each of them sending concurrently $N - 1$ RD-set messages to the other processors. The required sending time of $N - 1$ RD-set messages from *one leaf* to the other leaves, is equal to $N + 2\log N - 2$ steps (according to the protocol of round 3) independently of the capacity of the fat-tree channels. However, by taking advantage of the high-capacity channels of the ideal fat-tree we achieve the delivering of all the $(N - 1) \cdot N$ RD-set messages in the same time ($N + 2\log N - 2$ steps), without any collisions or delays:

Proposition 3.5. *The delivery of all the corresponding messages during round* 3 *is performed without any collisions or delays, thus meeting the requirements of the area-universal fat-tree model given in* [14]. *The communication time T_{r3} needed for the execution of round* 3 *of our algorithm is equal to*

$$T_{r3} = N + 2\log N - 2 \quad \textit{RD-set message steps}. \tag{5}$$

Proof. By induction on the height of the tree (h):

1. It is valid for $h = 1$. In this case we have $N = 2$ (the trivial case of a binary tree with only two leaf nodes) and $T_{r3} = N + 2\log N - 2 = 2$ steps (obviously,

processor 1 sends to processor 2 and processor 2 sends to processor 1, in two steps and in parallel).

2. We assume that it is valid for $h = k$, which means that for a fat-tree of height k ($N = 2^k$), round 3 of our algorithm takes $N + 2\log N - 2 = 2^k + 2\log 2^k - 2$ steps.
3. We will prove that it is valid for $h = k + 1$ ($N = 2^{k+1}$):

A fat-tree of height $h = k+1$ contains two subtrees of height $h = k$ ($N = 2^k$). We name them T_{left} and T_{right}. For each one of them (according to the assumption) the time needed to exchange all the $2^k \cdot (2^{k-1})$ messages is equal to $2^k + 2\log 2^k - 2$. The messages that remain to be exchanged (according to the protocol of round 3 of our algorithm) are the ones that leave the 2^k leaves of T_{left} (2^k messages from each leaf) in the direction toward the 2^k leaves of T_{right} and vice versa. Actually T_{right} is the kth (and the last one) adjacent subtree of T_{left} and vice versa. We focus on the T_{left} subtree. Each one of its leaves has to send 2^k messages (one for each leaf of T_{right}). All the leaves of T_{left} send the messages concurrently according to the order implied by our protocol:

(a) The beginning can be placed at moment 2^k. At this moment each leaf of T_{left} has finished sending the $2^k - 1$ messages toward the other leaves of T_{left}.

(b) According to our protocol, at each of the following 2^k moments the 2^k leaves of T_{left} send concurrently 2^k messages. Following the same considerations with round 5 of our algorithm (disjoint paths), each such message set arrives at the root in $\log 2^{k+1}$ steps. Thus, the first set will arrive at the root at moment $2^k + \log 2^{k+1} - 1$. The other message sets follow in the same way (by pipelining) without delays. Thus, the last set will arrive at the root at moment $2^k + (2^k - 1) + \log 2^{k+1} - 1$.

(c) In the T_{left} subtree there are no collisions between these 2^k message sets and the $2^k - 1$ sets that have been exchanged internally. This happens because the latter are always in front of the former, and they change direction before arriving at the root. We have to remember here that all the links are bidirectional. So there is no problem when two messages use the same link concurrently, but in the opposite direction.

(d) The protocol of round 3 implies that the first message set (2^k messages) has 2^k different destinations (the leaves of T_{right}). Thus, following the same considerations with round 1 of our algorithm (disjoint paths), each set arrives at the leaves in $\log 2^{k+1}$ steps. Also our protocol guarantees (going right and circularly) that the following $2^k - 1$ message sets will have different directions too. Consequently, the delivery time of $\log 2^{k+1}$ steps is valid for all the message sets.

(e) There are no collisions between the above 2^k message sets (when they go down toward the leaves of T_{right}) and the messages that are exchanged internally in T_{right}. This happens because the last message set of the internal messages of T_{right} arrives at the root of T_{right} (switch node of level k) at moment $2^k + \log 2^k - 2$ (the total time needed for the internal exchange of messages minus the time needed for this last set to arrive at the leaves). However, the first set of the other 2^k messages (see (d)) arrives at the root of the whole tree (one level up) at moment $2^k + \log 2^{k+1} - 1$. In total there is a difference of three steps.

Consequently (from (a)–(e)) there are no collisions or delays, and the pipelining is not influenced in any case. Thus, we can simply add the times mentioned at (b) and (d) and complete the proof:

$$\begin{aligned} T_{\text{total}} &= 2^k + (2^k - 1) + \log 2^{k+1} - 1 + \log 2^{k+1} \\ &= 2^{(k+1)} + 2\log 2^{(k+1)} - 2 = N + 2\log N - 2. \end{aligned}$$

□

Finally, (a) based on the partial execution times given by Propositions 3.1–3.5 and (b) under the assumption that the query messages and the RD messages are almost of the same size (we can make the above assumption since queries of 10–20 terms and initial relevant documents sets of 10–20 documents is a usual case of the interactive text retrieval systems), we easily conclude the following lemma, which states the total execution time of our algorithm.

Lemma 3.6. *The total time needed for the execution of the general multiple-queries algorithm presented in section* 3.3.2, *is equal to*

$$T_{\text{total}} = 2N + 4\log N - 1 + N \cdot (T_{\text{s}} + T_{\text{lr}}) + 2 \cdot (N - 1) \cdot R. \tag{6}$$

Moreover, the corresponding amortized processing time for each query can also be directly obtained by dividing the total execution time (stated in the previous lemma) by the total number of queries N. Thus:

Lemma 3.7. *Execution of the general multiple-queries algorithm presented in Section* 3.3.2 *results in an amortized processing time per query equal to*

$$T_{\text{a}} = 2 + \frac{4\log N - 1}{N} + (T_{\text{s}} + T_{\text{lr}}) + 2 \cdot R. \tag{7}$$

Actually, in (7) the merging time is equal to $2 \cdot ((N-1)/N) \cdot R$, but for simplicity we keep only the $2 \cdot R$ factor (generally the difference is not worth telling, especially for large values of N).

4. Further Discussion on the Performance of the Algorithm

As is seen from Lemma 3.7, an almost perfect (expected) speed-up is achieved in respect to the amortized processing time per query, since:

1. The computational tasks (scoring and ranking) are parallelized almost perfectly in almost the same way as in [7].
2. The computation overhead due to merging the results is restricted to $O(R)$ time per query which is extremely small compared to the other computation tasks (e.g., the parameter R which represents the size of the relevant document sets is usually equal to 10–20 documents).
3. The total communication amortized time is almost constant and equal to 2 (especially for large values of the parameter N).

Considering the topology of our interconnection network (ideal fat-tree) and the heavy query load, we can say that our algorithm has a very efficient behavior. In the rest of this section we give a comparison of our multiple-queries algorithm with the single-query algorithm of [7].

4.1. *Comparison with the Single-Query Algorithm*

We consider the case of a complete binary tree as an interconnection network of N processors (the nodes of the intermediate levels of the tree are also working processors), where the arriving text queries are processed sequentially:

Lemma 4.1 (from [7]). *The total time needed for processing one text query using the binary-tree single-query algorithm is equal to*

$$T_{\text{tb}} = (2\log N + 2) + (T_{\text{s}} + T_{\text{lr}}) + 4R \cdot (\log N - 1). \tag{8}$$

The factor $2\log N + 2$ corresponds to the message steps required for bidirectional communication (query submission and collection of results, including the host), $(T_{\text{s}} + T_{\text{lr}})$ corresponds to the time required for the execution of scoring and ranking tasks, and $4R \cdot (\log N - 1)$ is the corresponding time required for merging. The fact that the internal nodes of the binary tree of [7] are also working processors (which is not the case for the fat-tree) does not actually affect the corresponding comparison because we suppose (as it is incorporated in (8)) a binary tree of height $h - 1$ ($h = \log N$ is the height of the fat-tree) which results in a binary tree with $N - 1$ working processors. The corresponding difference ($N - 1$ versus N) of one working processor influences the factor only a little, whereas it is actually insignificant for large values of N. The binary-tree single-query algorithm has been proved to be very efficient from both the theoretical and practical point of view. In Table 1 the reader can find the total execution times from Lemmas 3.7 and 4.1 and some result obtained by the replacement of R with 10, and for $N = 8$ and 512. The reader can see the obvious differences between the two algorithms, which increase for large N. In the single-query algorithm the communication and the merging times are increased logarithmically compared with N, while in the multiple-queries algorithm they remain almost constant.

We must also note that the execution of the binary-tree algorithm on a fat-tree would not increase its efficiency since its communication tasks are executed optimally even

Table 1. Examples for various values in (7) and (8).

	Messages	Computing	Merging
Fat-tree, multiple queries	$2 + (4\log N - 1)/N$	$T_{\text{s}} + T_{\text{lr}}$	$2R$
Binary tree, single query	$2\log N + 2$	$T_{\text{s}} + T_{\text{lr}}$	$4R(\log N - 1)$
Fat-tree, multiple queries, $R = 10$, $N = 8$	3,4	$T_{\text{s}} + T_{\text{lr}}$	20
Fat-tree, multiple queries, $R = 10$, $N = 512$	2,1	$T_{\text{s}} + T_{\text{lr}}$	20
Binary tree, single query, $R = 10$, $N = 8$	8	$T_{\text{s}} + T_{\text{lr}}$	80
Binary tree, single query, $R = 10$, $N = 512$	20	$T_{\text{s}} + T_{\text{lr}}$	320

on the binary tree. Obviously, it is impossible to restrict the communication overhead below $O(\log N)$ for any single-query processing algorithm over any binary-tree-based architecture (either a simple binary tree or an ideal fat-tree) . Thus, the above comparison between the two algorithms is completely fair although the ideal fat-tree is (generally) a more powerful architecture than the binary tree. Furthermore, it is obvious that the same improvements hold in the general case of $a \cdot N$ (multiples of N) queries by simply allowing the fat-tree algorithm to operate in a sequential phases by processing (concurrently) one set of N queries at each phase. Generally, as is shown above, the worth of using the fat-tree multiple-queries algorithm is substantial (concerning the amortized processing time per query) (a) when we have a big number ($a \cdot N$) of queries concurrently arriving to the system and (b) when we have to process a batch of ($a \cdot N$) queries in very low total processing times. In the following subsection we show that almost the same improvements hold also concerning the average waiting time per query (the time needed for extracting the response for each one of the $a \cdot N$ queries in average).

4.2. *Analysis of the Average Waiting Times*

Supposing that we have a number of $a \cdot N$ concurrently arriving queries, the correponding average waiting times per query for the binary-tree single-query and the fat-tree multiple-queries algorithms can easily be computed since we already know the waiting time for the first query in the first case (T_{tb}) and for the first N queries in the second case ($T_{\text{total}} = N \cdot T_{\text{a}}$). The following lemmas state formally the necessary computations and average waiting time results for both algorithms.

Lemma 4.2. *Given a total number of $a \cdot N$ concurrently arriving queries, the average waiting time per query, T_{tbw}, for the binary-tree single-query algorithm is equal to*

$$O(N \log N) \textit{ message steps} + \frac{(aN+1)}{2} \cdot (T_{\text{s}} + T_{\text{lr}}) \textit{ computation time} + O(N \log N) \textit{ merging time.}$$

Proof. It is easy for the reader to notice that the waiting time for the first query is T_{tb}, for the second query it becomes $2T_{\text{tb}}$, etc., till the last query, which has $aN \cdot T_{\text{tb}}$ waiting time. Consequently, by evaluating the average of the above waiting times we obtain (T_{tb} is taken from Lemma 4.1)

$$\begin{aligned} T_{\text{tbw}} &= \frac{1}{aN} \cdot (1 + 2 + \cdots + aN) \cdot T_{\text{tb}} = \frac{1}{aN} \cdot aN \cdot \frac{(aN+1)}{2} \cdot T_{\text{tb}} \\ &= \frac{(aN+1)}{2} \cdot T_{\text{tb}} \\ &= \frac{(aN+1)}{2} \cdot [(2 \log N + 2) + (T_{\text{s}} + T_{\text{lr}}) + 4 \cdot R \cdot (\log N - 1)] \\ &= O(N \log N) + \frac{(aN+1)}{2} \cdot (T_{\text{s}} + T_{\text{lr}}) + O(N \log N). \end{aligned}$$

□

Lemma 4.3. *Given a total number of $a \cdot N$ concurrently arriving queries, the average waiting time per query T_{tfw} for the fat-tree multiple-queries algorithm is equal to*

$$O(N + \log N) \textit{ message steps} + \frac{(aN + N)}{2} \cdot (T_s + T_{lr}) \textit{ computation time} + O(N) \textit{ merging time}.$$

Proof. In this case we can easily observe that the waiting time for each one of the first N queries is $N \cdot T_a$, for the second set of N queries it becomes $2N \cdot T_a$, etc., till the last N queries which have $aN \cdot T_a$ waiting time. Consequently, by taking the average, we obtain (T_a is taken from Lemma 3.7)

$$\begin{aligned} T_{\text{tfw}} &= \frac{1}{a} \cdot (1 + 2 + \cdots + a) \cdot NT_a = \frac{1}{a} \cdot a \cdot \frac{(a+1)}{2} \cdot NT_a = \frac{(aN + N)}{2} \cdot T_a \\ &= \frac{(aN + N)}{2} \cdot \left[\left(2 + \frac{(4 \log N - 1)}{N} \right) + (T_s + T_{lr}) + 2 \cdot R \right] \\ &= O(N + \log N) \; + \; \frac{(aN + N)}{2} \cdot (T_s + T_{lr}) + O(N). \end{aligned}$$

□

Based on the above results the following conclusions can be extracted:

1. There are significant improvements concerning the communication time and the merging time ($O(N + \log N)$ versus $O(N \log N)$ and $O(N)$ versus $O(N \log N)$, respectively) which are similar to the improvements concerning the amortized processing times ((7) and (8)).
2. There is a small overhead of almost $(N-1)/2 * (T_s + T_{lr})$ concerning the computation time of the fat-tree algorithm ($(aN + N)/2$ versus $(aN + 1)/2$). However, this overhead is rather insignificant in our case of a large number of concurrently arriving queries ($aN \gg N$). In this case the factor aN (concerning the computation times) and the improvements referred above (concerning the communication and the merging time) clearly dominate over the small computation overhead introduced.

Furthermore, the computational overhead tends to be significant only in the case of low values of a, e.g., for $a = 1$ the computational overhead of $(N-1)/2 * (T_s + T_{lr})$ could probably be comparable to the improvement in communication times during a real experiment. However, even in this case (only N concurrently arriving queries) we can overcome the arising overhead by submitting the N queries (into the fat-tree multiple-queries algorithm) in N/k sequential phases of k queries processed in each phase (where $k = 2^i | i = 1 \cdots \log N$). Consequently, the corresponding average waiting times become as follows (similar formulas are easily extracted for the general case of $a \cdot N$ queries):

$$T_{\text{tbw}} = O(N \log N) + \frac{(N+1)}{2} \cdot (T_s + T_{lr}) + O(N \log N),$$

$$T_{\text{tfw}} = O(N + \log N) + \frac{(N+k)}{2} \cdot (T_s + T_{lr}) + O(N).$$

As is also seen from the above formulas, even if we have only N concurrently arriving queries, the significant improvements in communication and merging times are

preserved at the same level. The arising computational overhead tends to be significantly restricted (N dominates over k) for large values of N and relatively small values of k.

The reader can notice that, considering the average waiting time per query, there is a tradeoff between the improvements in communication times and the arising computational overhead caused by the multiple-queries algorithm. This is due to the fact that the results (relevant documents sets) for each set of N queries are extracted at the end of processing all the N queries (not one by one, as is the case for the sequential execution of the single-query algorithm). In order to take advantage of the high-capacity channels of an ideal fat-tree structured network (toward the minimization of the total communication times), we have to pay something on the average waiting time.

It is really worth following this direction, since in the case of a heavy query load ($|X| = aN \gg N$) the corresponding tradeoff does not actually influence the total performance improvements provided by the multiple-queries algorithm over the single-query algorithm. In the case of smaller sets of queries the improvements still hold, although they are somewhat restricted due to the additional computational overhead. However, in the case of relatively very small sets of queries ($|X| \leq N$) the corresponding tradeoff would substantially influence the total performance and the two algorithms could be regarded as equivalent (assuming a straightforward extension of the multiple-queries algorithm for processing less than N queries).

5. Optimality and Extensions of the Algorithm

In this section we present some theorems and proofs stating the optimality of our algorithm considering the communication times needed for the execution of the corresponding communication tasks (rounds 1, 3, and 5 of the algorithm) in ideal fat-trees with $c(n) = n$. Also we present an appropriate generalization of the algorithm for a more general class of fat-trees where the desired communication optimality is also being preserved.

As is seen from (3)–(5) that the communication time needed for the execution of round 3 clearly dominates over the times needed for the execution of rounds 1 and 5. Moreover, it is easy for the reader to notice that the communication tasks implied by rounds 1 and 5 are executed optimally (in the sense of exact analysis). Considering round 5 the optimality is obvious, whereas considering round 1 (where each processor has to receive N different messages), the corresponding proof of optimality is rather trivial. Specifically, since the capacity of the channel connecting each processor to his parent is equal to 1, at least N message steps are required in any case in order for the N messages to cross this channel, plus another $\log N - 1$ steps for the first message to arrive at the processor's parent (in any case), plus one step for the host-to-root communication, thus making a total of $N + \log N$.

Consequently, the optimality of the communication protocol of round 3 directly implies the optimality of the whole communication task (rounds 1, 3, and 5) of the algorithm.

The communication task implied by round 3 of the algorithm can be stated in a more formal and simplified way as follows (similarly to the known all-to-all personalized routing problems):

- *What is the communication cost for the multiple concurrent exchanging of different messages among all the processors* (*each processor sends* $N-1$ *different messages of fixed size to all the others*) *of an ideal fat-tree of size* N *with capacity function* $c(n)=n$?

In the following we refer to the above problem as the Multiple Message Exchange (MME) problem and we prove that it is solved optimally by the communication protocol described in round 3 of our algorithm.

Theorem 5.1. *The concurrent exchanging of* $N\cdot(N-1)$ *different messages among all the processors of an ideal fat-tree with* $c(n)=n$ *cannot be performed in less than* $T_{\text{com}}=(N+2\log N-2)$ *communications steps. Consequently,* T_{com} *is a strict lower bound for the MME problem in ideal fat-trees with* $c(n)=n$.

Proof. Clearly, each processor p of one subtree (e.g., the left subtree T_{l}) of the fat-tree has to send $N/2$ messages (we call this set of messages M_1) to the $N/2$ processors of the other subtree (T_{r}). Moreover, each processor p has to send another $N/2-1$ messages (we call this set of messages M_2) to the other $N/2-1$ processors of its own subtree (T_{l}). The two alternative possibilities concerning the order of sending all the above messages are as follows:

1. Processor p sends the messages of both sets M_1 and M_2 mixed in such a way that the last message belongs to set M_1. This sending order includes the case of sending all the messages of set M_1 after all the messages of set M_2, which is followed by our algorithm. In any case we have a total of at least $N-1$ sending steps (for all the messages) plus the additional $2\log N-1$ steps required for the last message in order for it to arrive at its destination (a working processor of the other subtree). Thus, the total time is at least $T_{\text{com}}=N+2\log N-2$ message steps.
2. Processor p sends the messages of both sets M_1 and M_2 mixed in such a way that the last message belongs to set M_2. This sending order includes the case of sending all the messages of set M_2 after all the messages of set M_1. In this case, we point out that if some other processor (except p) follows the above sending order, then the total time required will again be at least $N+2\log N-2$ steps, independently of the sending order of p. Thus, in the following we examine the case in which all the processors follow the present sending order. This is succeeded by randomly spreading a number of the messages of type M_2 into the complete message sets of type M_1, which is the beginning instance in our proof.

We concentrate on the messages of the sets M_1 of all the processors of the left subtree (T_{l}). The reader can notice that the optimal time in which all these messages ($(N/2)^2$ in total) could arrive at the leaves of the right subtree (T_{r}) is $T_{\text{m}}=(N/2+2\log N-1)$ provided that all the processors of T_{l} continuously send such messages (of type M_1) during all the first $N/2$ moments. T_{m} is a lower bound for the above task because there are at least $N/2$ sending steps for each processor and $2\log N-1$ additional steps for the last message in order for it to arrive at its destination. Now, if some processor of T_{l} sends one message of set M_2 before the end of sending messages of type M_1 (at any random

moment), there will be at least one processor in the right subtree that will receive one M_1 message after the moment T_m. This happens because in that case the specific processor of T_l will be forced to send at least one message of type M_1 after the $N/2$ moment and this message will spend at least $2 \log N - 1$ steps in order to arrive at its destination.

Generally, if k such messages (of type M_2) are sent in total by all the processors of T_l before the end of sending the messages of type M_1, there will be at least k messages of type M_1 that will arrive at the processors of T_r after moment T_m. Beyond these k messages, the processors of T_r will have to wait for another $((N/2-1) \times (N/2)) - m$ messages of type M_2 that will be sent internally by the processors of T_r and will arrive after the moment T_m. However, the latter implies that the processors of T_r have sent m such messages before the end of sending of their own messages of type M_1. Consequently, there will be m messages of type M_1 that will arrive at the processors of T_l after the moment T_m. Moreover, there will be another $((N/2-1) \times (N/2)) - k$ messages of type M_2 that will arrive at the processors of T_l after the moment T_m.

In total, concerning the processors of the right subtree (T_r), there will be at least $S_1 = k + ((N/2-1) \times (N/2)) - m$ messages that are expected after the moment T_m, whereas, following the same considerations, there are also at least $S_2 = m + ((N/2 - 1) \times (N/2)) - k$ that are expected to arrive at the processors of the left subtree (T_l) after the moment T_m. These messages have to cross (either in T_l or in T_r) at least some (or all of them) of the $N/2$ unit-capacity links that connect the processors of the corresponding subtree to their parents. Thus, considering the right subtree (T_r), if $k \geq m$ a total of at least $(N/2 - 1)$ additional steps are required in the best case (by dividing S_1 with all the $N/2$ unit-capacity links). If we add them to the $T_m = (N/2 + 2 \log N - 1)$ previous steps, we conclude with a total of at least $T_{com} = (N + 2 \log N - 2)$ steps for the whole execution. On the other hand, if $k < m$ the above is not true for T_r, but it holds respectively for T_l. □

As an example, concerning the specific case (which initially looks as the most preferable solution) that all the "long-distance" messages of type M_1 are sent before all the messages of type M_2 (as opposed to our algorithm), we have $k = m = 0$, and all the $((N/2-1) \times (N/2))$ internally exchanged messages in each subtree will have to arrive at the leaves (at least to cross one unit-capacity link) after the moment T_m (due to the multiple collisions with the externally exchanged messages). Thus, at least $N/2 - 1$ additional moments are required for all of them in order to cross some (or all) of the $N/2$ unit-capacity links.

Consequently, since the communication protocol executed in round 3 of our algorithm achieves a communication time equal to $(N + 2 \log N - 2)$ steps, it is proved to be exactly optimal for ideal fat-trees with a capacity function $c(n) = n$.

Furthermore, the communication protocol implied by round 3 of the algorithm can be efficiently applied over the general class of fat-trees with the capacity function

$$c(n) = n/2^i | i = 0 \cdots \log N - 1.$$

Generally, the above general class of fat-trees includes all the fat-trees which start doubling their edge capacity from level i ($i = 0 \cdots \log N - 1$). Moreover, this class ranges from the simple binary trees ($i = \log N - 1, c(n) = 2$) to the general ideal fat-trees ($i = 0, c(n) = n$). Considering a fat-tree with the above general capacity function,

the communication protocol implied by round 3 of our algorithm can be generalized in the following way:

- Execute all steps implied by round 3 in the same order but introduce a delay of 2^i moments between the "sending" steps of each processor. Thus, each processor is allowed to send one message every 2^i moments.

Obviously, for $i = 0$ (ideal fat-trees with $c(n) = n$) all the processors send their messages continuously (one message at each moment) as is implied by round 3 of the original algorithm. Making the same considerations as in Section 3.3.3, Proposition 3.3, it is easy to prove that the above generalized communication protocol takes time $[2^i N + 2\log N - 2^i - 1] = O(2^i N)$ without any collisions or delays. This time ranges from $(N + 2\log N - 2) = O(N)$ for $i = 0$ (ideal fat-trees with $c(n) = n$) to $O(N^2)$ for $i = \log N - 1$ (simple binary trees). Furthermore, we can prove that this generalized communication protocol (efficiently solving the MME problem for all fat-trees with $c(n) = n/2^i$) is also optimal in the following sense:

Theorem 5.2. *The MME problem cannot be solved in time less than $O(2^i \cdot N)$ for that class of fat-trees with size N and capacity function $c(n) = n/2^i$.*

Proof. Clearly, considering the specific general class of fat-trees we can observe that the capacity of the root is always equal to $N/2^i$ and it is equally shared to its two children. Thus, there are $N/2^{i+1}$ links connecting the root to its right child. Also, it is obvious that there are $(N/2)^2$ messages which have to pass through the root from left to right (the messages sent by the $N/2$ processors of the left subtree toward the $N/2$ processors of the right subtree). Consequently, since there are only $N/2^{i+1}$ links connecting the root to its right child the total time needed for all the $(N/2)^2$ messages to pass through the root is at least $(N/2)^2/(N/2^{i+1}) = (2^i N)/2 = O(2^i N)$. □

The generalization of the communication protocol of round 3 that is proposed above is optimal, since it actually achieves $O(2^i N)$ time. Making the same considerations as above we can easily state another more general theorem that holds for any capacity function of a fat-tree of size N:

Theorem 5.3. *For any given fat-tree of size N, if the root capacity is equal to R, the MME problem cannot be solved in time less than $O(N^2/R)$.*

Proof. Again, there are $(N/2)^2$ messages that have to pass through the root from left to right, whereas there are $R/2$ links connecting the root to its right child. Thus the total time needed for all the $(N/2)^2$ messages to pass through the root is at least $(N/2)^2/(R/2) = O(N^2/R)$. □

The above theorems indicate a significant restriction (the root capacity) with respect to the optimal time that can be achieved for the MME problem on a fat-tree of any given capacity function. As a consequence, an important question arises concerning the

"utilization" of the total link capacity (total number of links in all levels) of the given fat-tree. In simple words, since there is a specific restriction concerning the optimal time that can be achieved, one has to examine what fraction of the total links of the fat-tree are actually occupied during the whole time of execution of the corresponding communication protocol. Trying to establish a relevant "utilization" measure we could say that

- we achieve perfect links utilization iff the product (namely L_a) of the total link capacity of the fat-tree with the time needed for exchanging all the messages (this product simply implies the total number of the "available" links of the fat-tree during the whole execution) is order-equal to the total number of links that all the messages have to cross during the execution of the protocol (namely L_n).

Consequently, an appropriate links-utilization measure ranging from 0 to 1 could be given by the fraction $U_L = L_n/L_a$. Concerning the optimal links utilization that can be achieved for the MME problem we can prove that:

Theorem 5.4. *The MME problem can be solved with perfect links utilization* ($U_L = L_n/L_a = O(N^2 \log N)/O(N^2 \log N)$) *in ideal fat-trees with* $c(n) = n$ *and the communication protocol described in round* 3 *of our algorithm achieves this utilization optimally.*

Proof. The reader can notice that $L_n = O(N^2 \log N)$. Specifically, there are $2(N/2)^2$ messages that have to pass through the root of the fat-tree ($(N/2)^2$ from the left subtree toward the right subtree and $(N/2)^2$ from the right toward the left). Each one of these $O(N^2)$ messages has to cross $2 \log N = O(\log N)$ links in order to arrive at its destination. Thus, we have a total of $L_n = O(N^2 \log N)$ links crossed during the whole execution. Also, the total links capacity of the ideal fat-tree (with $c(n) = n$) is equal to $2N \log N = O(N \log N)$ (we assume bidirectional links). Since the best time that can be achieved for the MME problem in an ideal fat-tree (Theorem 5.1) is $O(N)$ we have $L_a = O(N \log N) \times O(N) = O(N^2 \log N)$. That means that $L_a = L_n$ and the links utilization U_L is equal to 1 for ideal fat-trees with $c(n) = n$. □

On the other hand, the above theorem does not hold generally for all the fat-trees of the general capacity function $c(n) = n/2^i | i = 0 \cdots \log N - 1$. Specifically, the links utilization of the general fat-trees of that capacity function (with regard to the MME problem and Theorem 5.2) decreases with the increase of i and finally becomes equal to $U_L = L_a/L_n = O(N^2 \log N)/O(N^3)$ for the simple binary trees with $c(n) = 2$. In this case, the total links capacity of the simple binary tree is equal to $2N - 1 = O(N)$, the best time that can be achieved (Theorem 5.2) is equal to $O(N^2)$ (thus, $L_n = O(N^3)$) and L_a remains $O(N^2 \log N)$ as in ideal fat-trees. Thus, for the general class of fat-trees, the MME problem cannot be solved with perfect links utilization in all cases.

6. Experimental Results

In order to obtain realistic experimental evidence concerning the performance of the multiple-queries algorithm and its superiority to the single-query algorithm, we have performed two basic sets of experiments (amortized processing times in Table 2, and average waiting times in Table 3) over the GCel3/512 Parsytec parallel environment. Both the experiments have been performed over the well-known TIPSTER TREC WSJ ('90–'92) collections (almost 75,000 documents of the *Wall Street Journal* of almost 250 MB) using the TREC 251–300 user topics (description fields). Also in both the experiments we have assumed relevant documents' sets of size 20 ($R = 20$), whereas we have used the 251–300 TREC topics multiple times (iteratively) when needed (e.g., for $N = 128$ we need 128 user queries in the first experiment, whereas in the second experiment we need much more).

The TIPSTER collections (a total of almost 3 GB of SGML text data) have been regarded as the most reliable large-scale collection data in recent information retrieval research. They have been widely used during the last 5 years and they are the official data used for the purposes of the well-known annual TREC conferences (sponsored by the National Institute of Standards and Technology of the U.S.A., see [11]).

Our experimental efforts have been based on the efficient simulation of an ideal fat-tree over the two-dimensional mesh-structured network of a Parsytec parallel machine, via the specific embedding method presented in the Appendix. This embedding offers a reliable simulation of the high-capacity channels of the ideal fat-tree, whereas it preserves the same key characteristics of the corresponding embedding of a binary tree over a two-dimensional mesh (see [7] and [17]). Consequently, if we omit (as is the case in our two basic experiments) the interswitch-node communication times of the fat-tree embedding, then we get a reliable platform for the experimental comparison between the two algorithms. Theoretically, the above statement means that we concentrate on (7) of Section 3.3.3, omitting the result of (10) of the Appendix.

Note that using the specific method, the maximum size of an efficiently embedded fat-tree over the GCel3/512 Parsytec machine is equal to 128 (this is the reason we do not present measurements for more than 128 processors in Tables 2 and 3). Also, the measurements for the binary-tree algorithm have been performed using the pre-existing PFIRE system [7], [17] and its recent extensions [16], and the fat-tree embedding (integrated with PFIREs IR features and TREC WSJ collections) has been implemented as a separate system.

In Table 2 the amortized processing times (R_p, in seconds) as well as the corresponding achieved utilization values (U_p, as a percentage) are presented for both algorithms. We present measurements for a varying number of processors (16, 32, 64, 128) and for a varying collection size ($\frac{1}{4}$ WSJ, $\frac{1}{2}$ WSJ, and full WSJ collection, by appropriately dividing the total number—almost 75,000—of the WSJ documents). The utilization measure is defined as the fraction $U_p = S_p/P$, where P is the number of processors (N in our case) and S_p is the well-known speed-up measure (the time needed for execution in one processor divided by the time needed for execution in P processors). In simple words the U_p measure implies the fraction of the maximum possible speed-up (equal to P) that is achieved in each case.

Table 2. Amortized processing times.

	Fat-tree, multiple queries						Binary tree, single query								
	$\frac{1}{4}$ WSJ		$\frac{1}{2}$ WSJ		WSJ		$\frac{1}{4}$ WSJ		$\frac{1}{2}$ WSJ		WSJ		Improved R_p (%)		
N	R_p	U_p	R_p	U_p	R_p	U_p	R_p	U_p	R_p	U_p	R_p	U_p	$\frac{1}{4}$ WSJ	$\frac{1}{2}$ WSJ	WSJ
16	0.44	84	—	—	—	—	0.57	81	—	—	—	—	22	—	—
32	0.25	81	0.40	88	—	—	0.39	72	0.52	81	—	—	36	23	—
64	0.14	76	0.26	86	0.38	89	0.26	59	0.37	73	0.50	84	45	31	24
128	0.081	74	0.15	85	0.29	93	0.17	42	0.25	61	0.39	86	53	40	26

The following conclusions can be directly extracted from Table 2 (the "—" in some cells means that the corresponding $1/x$ WSJ collection does not fit in the main memory of the processors of the machine).

1. The algorithm for multiple queries over fat-trees gives an almost perfect speed-up (e.g., 93% utilization on the full WSJ collection and 128 processors). This is due to (a) the small communication overhead and (b) the large memory capacity of the machine (2 GB). The single-query algorithm also gives quite good results (e.g., 86%) for the utilization measure (due mainly to (b)), but clearly much worse than the one offered by the multiple-queries algorithm.
2. Furthermore, the above difference (from 86% to 93%) between the two algorithms is increased with the increase of N and the decrease of the collection size. For the single-query algorithm this is due to the considerable effect of the logarithmic communication times (Amdahl's law, see [2]), since the computation tasks are getting smaller.
3. For the multiple-queries algorithm and regarding the total response times (see the percentage improvements in the last three columns), the same conclusions can be extracted. As an example we can refer to the case of the $\frac{1}{2}$ WSJ collection on 128 processors, where the utilization is 85% and 61% respectively for the two algorithms, and the achieved improvement in the response time (R_p) is increased to 40%.
4. The response times (R_p) are also very good. The better case with the multiple-queries algorithm is that of the full WSJ collection on 128 processors with response time 0.29 s. The single-query algorithm also gives very good results (0.39 s under the above conditions).

The reader may have a more clear view of the above conclusions by observing the utilization curves (with use of the $\frac{1}{4}$ WSJ collection) presented in Figure 4. As an example the fact that the "distance" between the two curves increases with the increase of N denotes that the improvements offered by the fat-tree algorithm increase with the increase of the number of processors.

As an overall conclusion (from Table 2) the experimental performance of the fat-tree multiple-queries algorithm (considering the amortized processing times) is substantially better than the one offered by the single-query binary-tree algorithm.

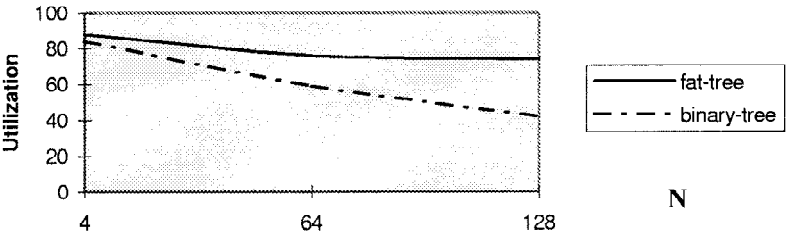


Fig. 4. Utilization curves for the $\frac{1}{4}$ WSJ collection.

In Table 3 the average waiting times (in seconds) are presented for both algorithms. We present measurements for a varying number of processors (16, 32, 64, 128) and concurrently submitted user queries. We have used varying values of a ($a = 5$, 10, 100, with regard to the context of Section 4.2) which result in substantially large numbers of concurrently arriving queries (e.g., for $N = 64$ we have 320 queries for $a = 5$, 640 queries for $a = 10$, and 6400 queries for $a = 100$). Note also that we have not used the same collection size for all values of N. Instead, we have used a collection size for each value of N that appropriately fills the total memory capacity of the N processors with data in each case (e.g., for $N = 64$ we have used the full WSJ collection, for $N = 32$ the $\frac{1}{4}$ WSJ, for $N = 16$ the $\frac{1}{4}$ WSJ and for $N = 128$ a simulated $2 \times$ WSJ collection). The following conclusions can be directly extracted from Table 3:

1. The average query waiting times offered by the fat-tree algorithm are better than the ones offered by the binary-tree algorithm even for a small value of a ($a = 5$, improvements up to 9.4%). The actual meaning of the above statement is that even a small value of a is enough in order to have an improvement in the average waiting time. By computing the corresponding average waiting times out of the values of Table 2, the reader can notice that for $a = 1$ the fat-tree algorithm would not be better than the binary-tree algorithm.
2. The corresponding improvements for greater values of a are substantially increased and they tend to be equivalent (e.g., for $a = 100$) to the improvements of the amortized processing times (Table 2). The reader can easily validate the above statement by comparing some percent improvements values of Table 3 with the corresponding values of Table 2, provided that the values of Table 2 correspond to cases of full-of-data processors (e.g., for $N = 64$ and full WSJ collection, we have 24% improvement from Table 2 and 23% from Table 3 with $a = 100$).

Table 3. Average waiting times.

	Fat-tree, multiple queries			Binary tree, single query			Improved R_p (%)		
N	$a = 5$	$a = 10$	$a = 100$	$a = 5$	$a = 10$	$a = 100$	$a = 5$	$a = 10$	$a = 100$
16	21.2	38.7	362.2	22.9	45.4	450	7.4	15	19.5
32	38.5	70.3	657.6	41.7	84.1	832.5	7.8	15	21
64	73	133	1241	79.5	158.1	1612	8.3	16	23
128	133.2	249	2312	147	298	3026	9.4	16.5	23.6

Table 4. Amortized processing times (full fat-tree embedding).

	Fat-tree, multiple queries						Binary tree, single query								
	$\frac{1}{4}$ WSJ		$\frac{1}{2}$ WSJ		WSJ		$\frac{1}{4}$ WSJ		$\frac{1}{2}$ WSJ		WSJ		Improved R_p (%)		
N	R_p	U_p	R_p	U_p	R_p	U_p	R_p	U_p	R_p	U_p	R_p	U_p	$\frac{1}{4}$ WSJ	$\frac{1}{2}$ WSJ	WSJ
16	0.47	83	—	—	—	—	0.57	81	—	—	—	—	18	—	—
32	0.33	77	0.46	84	—	—	0.39	72	0.52	81	—	—	16	11	—
64	0.23	62	0.34	75	0.48	85	0.26	59	0.37	73	0.50	84	12	8.5	4
128	0.158	43	0.237	62	0.38	87	0.17	42	0.25	61	0.39	86	7	5	2.5

As an overall conclusion from Table 3, we can say that the corresponding experimental results validate the theoretical analysis of Section 4.2, since the fat-tree multiple-queries algorithm can operate—in practice—very efficiently and fully interactively under a heavy query load.

Generally, the objective of our basic experimental efforts was a fair and reliable comparison between the fat-tree and the binary-tree algorithm. All the results would be much better if we had a real fat-tree as a physical interconnection network.

Also, if we had encountered the interswitch-node communication times of the fat-tree embedding, we would still have considerable improvements for fat-trees up to (but not more than) 128 nodes. The latter approach leads to a rather unfair comparison between the two algorithms, but makes a "real" and almost complete embedded fat-tree text retrieval system over the Parsytec GCel3/512 machine. On the contrary, as is seen from Tables 1–3, under a fair comparison between the two algorithms the corresponding improvements offered by the fat-tree algorithm increase substantially with the increase of the number of processors involved in the experiments.

Corresponding experiments (with the interswitch-node communication times of the fat-tree embedding encountered too) have already been performed in [16], using a straightforward extension of the communication protocol of round 3 of the algorithm. Some of the above experiments are briefly presented in Table 4. In this case, we have to incorporate the result of (10) of the Appendix and concentrate on the results of Table 6 instead of the results presented in Table 1. As is also pointed out in the Appendix, the large number of links of an ideal fat-tree ($O(N \log N)$ compared with the $O(N)$ links of the two-dimensional mesh) leads to undesirable communication overheads (given by (10) of the Appendix) for large values of N. This prevents the efficient implementation of any fat-tree embedding over a two-dimensional mesh of the same size.

As is seen from Table 4 (which is structured in the same way as Table 2), significant improvements still hold mainly for a small number of processors (16% and 18% for 32 and 16 processors, respectively). These improvements decrease significantly with the increase of the collection size, due to the same reasons mentioned above for the results of Table 1. The main difference between Tables 2 and 4 (as was expected due to the extra communication overheads caused by the fat-tree embedding) is the substantial decrease of the corresponding improvements as the number of processors increase (2.5%–7% for 128 processors). The above decrease would be even more obvious if we could simulate fat-trees of a greater size.

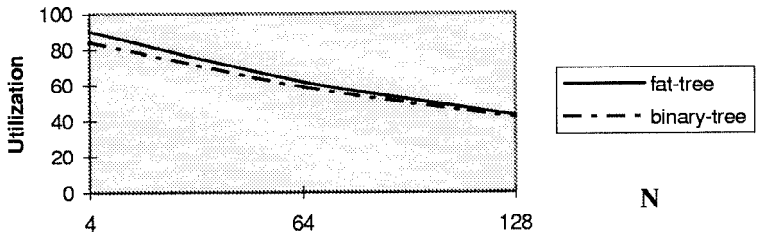


Fig. 5. Utilization curves for the $\frac{1}{4}$ WSJ collection (full fat-tree embedding).

Additionally, in order to obtain a more clear view of the above differences we present in Figure 5 the corresponding utilization curves for both algorithms when the full version of the fat-tree embedding is used. The reader may easily notice how the "distance" between the two curves decreases with the increase of N, as opposed to Figure 4 where the utilization improvements increase progressively with the increase of N.

An even more realistic approach to a complete embedded fat-tree text retrieval system over the specific machine implies the use of all the internal switch nodes of the fat-tree embedding as working processors. However, the corresponding design and implementation would demand a substantially different communication protocol concerning the communication task of round 3. We think it exceeds the purposes of the present work, which concentrates on the worth of use of "ideal" fat-trees on multiple text queries processing systems, but it is of high priority in our future work.

Finally speaking about the experiments presented in this section, we have to note the following remarks:

1. The way that the indexed document-term vectors are distributed over the multiple processors usually affects significantly the efficiency of a VSM-based parallelization due to the different sizes of those vectors. As is mentioned in Section 3.3.1, we have assumed an almost perfect distribution toward the direction of the best possible parallelization of the scoring and ranking tasks of the algorithm over the multiple processors of the machine. We have achieved such a distribution via an off-line data sharing and balancing scheme which is described in detail in [16]. This data-distribution scheme is mainly based on sorting all the document-term vectors according to their sizes, and finally leads to almost perfect results for text collections containing documents of "near" differences in their sizes (such as the WSJ collection which is used in our experiments).
2. In order to compute all the speed-up and utilization values, the total execution time of the algorithm on one processor had to be computed. However, in most of the previous experiments it could not be computed directly, since the total collection data used cannot fit the primary storage of one processor (4 MB) of the GCel3/512 machine. Therefore, we followed the alternative and reliable solution of (a) computing the total execution time including the necessary disk I/O access times and (b) subtracting those separately measured disk access times from the total execution time.

7. Conclusions and Further Work

Our work demonstrates the value of using an ideal fat-tree topology as the interconnection network of a multiprocessor system for parallel text retrieval using the Vector Space Model. We have presented and evaluated (both analytically and experimentally) a high-performance parallel multiple-queries algorithm for fat-tree structured networks, which is strongly recommended for text retrieval systems with a heavy query load. Moreover, our multiple text queries processing method is quite general and it could be efficiently applied over any other high-capacity network of processors or workstations (with the use of suitable communication protocols for each case). Specifically

1. We have achieved almost maximum parallelization by restricting the amortized (per query) communication time to an almost constant number of messages independently of the number of processors. The amortized required merging time is also restricted to $O(R)$ per query and almost the same achievements also hold for the average query waiting times.
2. We have shown that the delivery times achieved for all the communication tasks of our algorithm are optimal for a general class of ideal fat-trees.
3. As is seen from both the analytical and experimental comparison with the binary-tree single-query algorithm, our algorithm could be ideal assuming that we are given an ideal fat-tree as the underlying physical interconnection network.
4. We have efficiently applied our algorithm on the two-dimensional mesh-structured network of the GCel Parsytec machine via a quite efficient embedding of ideal fat-tree topologies on two-dimensional mesh-structured networks. This results in (a) almost perfect experimental results when the reduced version of the fat-tree embedding is used (omitting the interswitch-node communication times) and (b) quite satisfactory experimental results for fat-trees of reasonable size (up to 64 or 128 processors) when the full version of the fat-tree embedding is used.

The possible use of other complex and high-capacity interconnection models (e.g., hypercubes, butterflies) for parallel multiple text query processing, remains to be studied and compared with the present ideal fat-tree-oriented processing techniques. Also, the integrated implementation of our fat-tree multiple-queries approach under the additional assumption that the internal switch nodes of the fat-tree embedding serve also as working processors (with the use of a different, suitable for this case, all-to-all communication protocol) will be considered in our future work.

Appendix. Presentation and Analysis of the Embedding of the Ideal Fat-Tree Topology over a Two-Dimensional Mesh-Structured Parallel Machine

In the following we present an efficient embedding of an ideal fat-tree over a two-dimensional mesh-structured parallel machine. A realistic application of our method has been developed over the two-dimensional mesh network of the GCel3/512 Parsytec machine. In this case the feature of installing virtual links (see [7], [17], and [28])

given by Parsytec is appropriately used and leads to an efficient implementation of the multiple-queries algorithm presented above. Furthermore, the application of this specific embedding over the Parsytec machine served as the basis for the extraction of all the experimental results presented in Section 6.

The most important problem of such an embedding is the realization of the fat-tree high-capacity channels (parallel links from node to node). We have to point out that an ideal fat-tree of size N, has $N-1$ internal nodes (switch nodes) in $\log N$ intermediate levels. Each of these nodes has the same number of links (direction downward) with the number of leaves of the corresponding subtree (e.g., for a tree of size N and $\log N$ levels, each node on level $k|k = 1 \cdots \log N$ has capacity equal to 2^k). Moreover, there are $2^{\log N - k}$ internal nodes on level k. In total the capacity channels of each level is $2^{\log N - k} \cdot 2^k = N$. Thus, we have $N \log N$ bidirectional links in the whole fat-tree, while there are $N-1$ switch nodes. On the contrary, a mesh of $n \times m = N$ nodes provides only $(n \times (m-1)) + (m \times (n-1)) = 2(n \times m) - n - m = 2N - (n+m) = O(N)$ bidirectional links.

Therefore, in order to simulate a switch node of the fat-tree we must use more than one mesh node. More specifically, since each switch node provides 2^k parallel links (direction downward) and the mesh node gives only four communication links, we have to use $2^k/4 = 2^{k-2}$ mesh nodes for each switch node. Moreover, since each switch node provides another 2^k links (with direction upward), we need another 2^{k-2} nodes, and that means we in total need 2^{k-1} mesh nodes for each switch node of the k level. Considering the above, we need (for all the switch nodes of the fat-tree)

$$\sum_{k=1}^{\log N} (2^{k-1} \times 2^{\log N - k}) = (N \log N)/2 \text{ nodes}$$

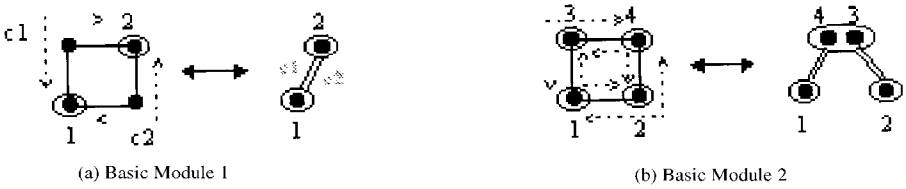
plus N nodes which will be the leaves of the fat-tree. The reader can find an example for $N = 16$ in Figure 7(b). Table 5 gives the relation between N and the required number of mesh nodes for some specific values of N (plus the required number of nodes needed for the embedding of a simple binary tree).

In the case of the Parsytec machine ($32 \times 16 = 512$ processors) we were able to simulate a fat-tree up to a size of $N = 64$ processors. This is a valuable result but the most important is to achieve an efficient simulation with reasonable communication times and a high level of speed-up for the multiple-queries algorithm.

Moreover (before the detailed presentation of our embedding technique), we have to note that the specific method has been chosen as the most appropriate for small fat-trees that fit in the GCel3/512 Parsytec machine. Considering more general cases of two-dimensional mesh-structured networks, perhaps other decomposition schemes (e.g., a butterfly scheme) could be more efficient.

Table 5. Mesh nodes required for binary tree and fat-tree embeddings.

N	4	8	16	32	64	128
Number of nodes of simulated binary tree on a mesh	7	15	31	63	127	255
Number of nodes of the simulated fat-tree on a mesh	8	20	48	112	256	576



(a) Basic Module 1 (b) Basic Module 2

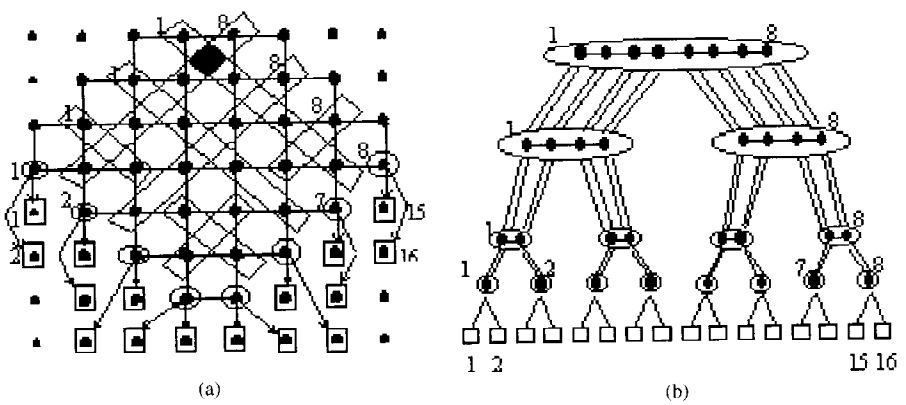
Fig. 6. The basic modules for the fat-tree embedding.

Specifically, our simulation is based on the following mappings:

1. A 2×2 grid can simulate (bidirectionally) two parallel links (or a switch node of capacity 2) in two steps, as is indicated in Figure 6(a).
2. With the same method, a 2×2 grid can simulate (bidirectionally) four parallel links among four nodes in two steps, as is indicated in Figure 6(b).

A careful reader can note that the above scheme has no conflicts (we assume bi-directional links), while there are two other remaining links for each mesh node for the rest of the communication needs of the switch node (e.g., upward for nodes 3 and 4 and downward for nodes 1 and 2).

Until now, we have specified the basic building blocks of our simulation. Dividing the fat-tree into the above blocks and implementing the mappings, we can take the final simulation (e.g., see Figure 7 for $N = 16$). This mapping shows a high utilization of the mesh links for the implementation of the switch nodes of the fat-tree. The reader can find a more comprehensive generalized description of our method at the end of this Appendix. Since the generalization of our method is not well-suited for the Parsytec machine (for values of N greater than 16), we use a modular construction for these values of N, in order for the simulation key characteristics to be preserved (e.g., for $N = 32$, we use two 8×8 grids, using the free nodes for the root switch node). This modular construction



(a) (b)

Fig. 7. Simulation of a fat-tree ($N = 16$) on the 8×8 two-dimensional grid.

finally allows the simulation of fat-trees up to 128 nodes over the two-dimensional mesh network of the GCel3/512 machine.

We point out the following conclusions:

1. The communications between the switch nodes of the fat-tree always take two steps of the mesh network. During each communication level only one of the 2^{k-1} mesh nodes that represent one switch node takes part. Thus, the size of all the N disjoint paths that connect the root to the leaves remains equal to $\log N$.
2. The communications that involve the leaves can also be completed within two steps.
3. Comparing the above simulation with the simple binary tree communications (see [7] and [17]), we can say that, in the latter case, we also need one or two physical steps (jumps) on the mesh (for larger values of N, the number of these steps is increased).

On the other hand, the main disadvantage of our simulation method is the required communication among the mesh nodes which compose a specific switch node. We give the analysis for these extra messages for the multiple-queries algorithm:

1. The reader must notice that the above simulation scheme shows a compatible behavior with the synchronization (e.g., pipelining) of the multiple-queries algorithm. In order to simulate the execution of the algorithm, we just have to add (into the pipelining) the required communication steps for the interswitch-node message passing, and keep the rest of the synchronization unmodified.
2. Assuming that the nodes of the root of the fat-tree can serve as the host of the system, we can say that there is no need for interswitch-node communication during rounds 1 and 5. This happens, since the simulation gives N disjoint paths between the leaves and the root of the fat-tree, each one having a size of $\log N$ intermediate nodes.
3. On the contrary, these communications take place during round 3 of the algorithm and for $k = 2 \cdots \log N$ (numbering from leaves to the root). This is the case when a message is sent, e.g., from a left subtree to the right subtree of a switch node of the fat-tree. This message is sent upward first by the leaves, passes the mesh nodes of the (local) root of the subtree, and finally it is sent downward to the leaves of the right subtree. Clearly, the required time for the above communication and for level k is much less than the corresponding time for level $k + 1$. For this reason we have to compute the total number of the exchanged messages for $k = \log N$ (messages passed from the root of the fat-tree):

Lemma A.1. *The total number of the additional communication messages introduced to the multiple queries algorithm by our fat-tree embedding method is dominated by the number of messages M passed from the root of the fat-tree, which is equal to*

$$M = 3N^2/16 - N/4. \tag{9}$$

Proof. There are $N/2$ nodes in the root-switch node of the fat-tree (e.g., see Figure 7 for $N = 16$). At each moment there are $N/2$ messages (according to our algorithm) arriving

from the right subtree at the $N/4$ right-positioned nodes ($N/4+1, \ldots, N/2$). Each of these messages have to change direction (toward the leaves) using the appropriate left-positioned node (the one that resides on the proper path). Totally, there are $N/2$ such message sets, each one of $N/2$ messages. Also, the same happens for the messages arriving from the left subtree to the $N/4$ left-positioned nodes of the root. However, since the execution is exactly the same, we focus on the case of the right subtree.

At each moment these $N/2$ messages arrive by couples at the $N/4$ right-positioned nodes. Concerning the first set each couple has to pass to the mapping left-positioned node ($i - N/4$, where i is the number of the right-positioned node). For each one of the following message sets, each couple has to pass to the next left-positioned node (with respect to the mapping node), etc. The delivery time for exchange of half ($N/4$) of the messages of each message set is equal to the delivery time for one message, e.g., for the first set the delivery time is $N/2 - N/4 = N/4$ message steps. This happens because the exchange can be executed in parallel. Moreover (see Figure 7), the mesh nodes that represent the root-switch node in our simulation can communicate with each other via two parallel links (in two parallel steps). Thus, the same delivery time is valid for each couple of messages (at each moment the two messages of each couple are exchanged in parallel).

Consequently, concerning, e.g., the first message set, all the $N/2$ messages ($N/4$ couples) can be exchanged in $N/2-N/4 = N/4$ steps. For each one of the next $N/2-1$ sets, the only difference is that the longest path (between one right-positioned node and the corresponding left-positioned node) grows by one (for each couple of sets). Actually, it becomes $N/2 - r$ ($r = 1 \cdots N/4$). This happens in the same way for each couple of sets. Thus, we have a total of

$$M = 2\sum_{r=1}^{N/4} N/2 - r = 2N^2/8 - 2N/4(N/4+1)/2 = 3N^2/16 - N/4. \qquad \square$$

Consequently, the amortized number of messages per query (M divided by N) is equal to

$$M_{\mathrm{a}} = 3N/16 - 1/4. \tag{10}$$

The result of (10) must then be added to the communication time of (7) in order to obtain the total amortized communication time needed for the execution of the multiple-queries algorithm using the specific embedding.

Table 6 demonstrates the total results (in the same way as Table 1 of Section 4.1) also giving a short comparison with the single-query binary-tree algorithm. The reader can see that for $N = 8$ and up to the value of 64 (small fat-trees that fit in the GCel Parsytec machine), the multiple-queries algorithm is more efficient. For greater N (e.g., $N = 512$) the binary-tree single-query algorithm presents better communication behavior. Totally (communication + retrieval + merging time) in this case, the performance of the two algorithms seems to be similar (this is due to the efficient merging times of the multiple-queries algorithm). For even greater N, the communication overhead of the multiple-queries algorithm is further increased, while the behavior of the single-query algorithm remains the same.

In the following, we present our detailed general method for embedding a fat-tree of size N on a two-dimensional mesh-structured network:

Table 6. Examples for various values in total time for both algorithms.

	Messages	Computing	Merging
Fat-tree, multiple queries	$2 + (4\log N - 1)/N + 3N/16 - 1/4$	$T_s + T_{lr}$	$2R$
Binary tree, single query	$2\log N + 2$	$T_s + T_{lr}$	$4R(\log N - 1)$
Fat-tree, multiple queries, $R = 10$, $N = 8$	4 6	$T_s + T_{lr}$	20
Fat-tree, multiple queries, $R = 10$, $N = 16$	5 7	$T_s + T_{lr}$	20
Fat-tree, multiple queries, $R = 10$, $N = 32$	8 5	$T_s + T_{lr}$	20
Fat-tree, multiple queries, $R = 10$, $N = 64$	14	$T_s + T_{lr}$	20
Fat-tree, multiple queries, $R = 10$, $N = 512$	98	$T_s + T_{lr}$	20
Binary tree, single query, $R = 10$, $N = 8$	8	$T_s + T_{lr}$	80
Binary tree, single query, $R = 10$, $N = 16$	10	$T_s + T_{lr}$	120
Binary tree, single query, $R = 10$, $N = 32$	12	$T_s + T_{lr}$	160
Binary tree, single query, $R = 10$, $N = 64$	14	$T_s + T_{lr}$	200
Binary tree, single query, $R = 10$, $N = 512$	20	$T_s + T_{lr}$	320

A fat-tree of size N occupies a mesh of dimensions $(N/4 + \log N) \times N/2$, where the rows are numbered from 1 to $N/4 + \log N$ and the columns are numbered from 1 to $N/2$. Then the fat-tree nodes are mapped on the two-dimensional mesh as described below:

1. The nodes of level $k = 1$ (in this level, each fat-tree switch node is represented by only one mesh node) are positioned in the following way: If we number them from 1 to $N/2$ (from left to right), then each node n ($n = 1 \cdots N/2$) occupies the mesh position $[N/4 + 2 - n, n]$ if $n \in (1 \cdots N/4)$ or the position $[n - 3, n]$ if $n \in (N/4 + 1 \cdots N/2)$.
2. For each level of the fat-tree, there are always $N/2$ mesh nodes (that represent all the switch nodes of this fat-tree level). If we number $1 \cdots N/2$ of these nodes of each fat-tree level, we obtain a 1–1 relation between the mesh nodes of each level. Thus, if n is one of the nodes of the first level ($k = 1$) with mesh position $[n_1, n_2]$, then the corresponding node of level k ($k = 2 \cdots \log N$) occupies the mesh position $[n_1 + (k - 1), n_2 + (k - 1)]$ if $n \in (1 \cdots N/4)$ or the position $[n_1 + (k - 1), n_2 - (k - 1)]$ if $n \in (N/4 + 1 \cdots N/2)$.
3. Concerning the leaves, there are two such leaf nodes for each node n of the first level of the fat-tree. If node n occupies the mesh position $[n_1, n_2]$, then its two leaves occupy the mesh positions $[n_1 - 1, n_2]$ and $[n_1 - 2, n_2]$ for all $n \in (1 \cdots N/2)$ except the following cases:
   if $n = N/4$ the two leaf nodes occupy the positions $[n_1 - 1, n_2]$ and $[n_1 - 1, n_2 - 1]$,
   if $n = N/4 - 1$ the two leaf nodes occupy the positions $[n_1 - 1, n_2]$ and $[n_1 - 2, n_2 - 1]$,
   if $n = N/4 + 1$ the two leaf nodes occupy the positions $[n_1 - 1, n_2]$ and $[n_1 - 1, n_2 + 1]$,
   if $n = N/4 + 2$ the two leaf nodes occupy the positions $[n_1 - 1, n_2]$ and $[n_1 - 2, n_2 + 1]$. □

References

[1] I. J. Aalbersberg and F. Sijstermans, High-Quality and High-Performance Full-Text Document Retrieval: The Parallel Infoguide System, *Proc*. 1*st International Conference on Parallel and Distributed Information Systems*, *PDIS* '91, Miami Beach, FL 1991.

[2] G. Amdahl, The Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities, *AFIPS Conference Proceedings*, Vol. 30, pp. 483–485.

[3] G. Billardi and P. Bay, An Area Lower Bound for a class of Fat-Trees, *Proc*. 4*th European Symposium on Algorithms*, *ESA* '94, 1994.

[4] G. Billardi, B. Codenoti, G. Del Corso, C. Pinotti, and G. Resta, Broadcast and Other Primitives Operations on Fat-Trees, *Proc*. *Europar* '97, pp. 196–207, Lectures Notes in Computer Science 1300, Springer-Verlag, Berlin, 1997.

[5] J. Cringean, R. England, G. Manson, and P. Willett, Parallel Text Searching in Serial Files Using a Processor Farm, *Proc*. *ACM SIGIR* '90, pp. 429–452, 1990.

[6] J. Cringean, M. Lynch, G. Manson, and P.Willett, Best Match Searching in Document Retrieval Systems Using Transputer Networks, British Library Research and Development Department, London, 1989.

[7] P. Efraimidis, C. Glymidakis, B. Mamalis, P. Spirakis, and B. Tampakas, Parallel Text Retrieval on a High Performance Supercomputer Using the Vector Space Model, *Proc*. *ACM SIGIR* '95 *Conference*, Seattle, July 9–13, pp. 58–66, 1995.

[8] M. Flynn, Some Computer Organizations and Their Effectiveness, *IEEE Transactions on Computers*, Vol. 21, pp. 948–960.

[9] O. Frieder and H. T. Siegelmann, On the Allocation of Documents in Multiprocessor Information Retrieval Systems, *Proc*. *ACM SIGIR* '91, pp. 230–239, 1991.

[10] F. Grandi, P. Tiberio, and P. Zezula, Frame Sliced Partitioned Parallel Signature Files, *Proc*. *ACM SIGIR* '92, pp. 286–297, June 1992.

[11] D. Harman, Overview of the Fourth Text Retrieval Conference, *Proc*. *Fourth Text Retrieval Conference*, *TREC* '95, November 25–27, Gaithersburg, MD, pp. 1–20, National Institute of Standards and Technology, Special Publication, 1995. (Electronic proceedings at http://www-nlpir.nist.gov/TREC.)

[12] M. Lafazanis, B. Mamalis, P. Spirakis, B. Tampakas, and A. Tsakalidis, FIRE: A Flexible Tool for Efficient Information Retrieval, *Proc*. *RIAO* '94 *Conference*, New York, October 11–13, pp. 132–140, 1994 (accepted in prototyped demonstrations).

[13] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures*: *Arrays*, *Trees and Hypercubes*, Morgan Kaufmann, San Mateo, CA, 1992.

[14] C. Leiserson, Universal Networks for Hardware-Efficient Supercomputing, *IEEE Transactions on Computers*, Vol. 34, No. 10, 1985.

[15] C. Leiserson, Z. Abuhamdeh, C. Douglas, C. Feynman, M. Ganmukhi, J. Hill, W. Hillis, C. Kuszmaul, M. A. St. Pierre, D. Wells, M. Wong, S. Yang, and R. Zak, The Network Architecture of the Connection Machine CM-5, *Proc*. 4*th ACM Symposium on Parallel Algorithms and Architectures*, pp. 272–285, July 1992.

[16] B. Mamalis, P. Spirakis, and B. Tampakas, Parallel Techniques for Efficient Searching over Large Text Collections, *Proc*. *Fifth Text Retrieval Conference*, *TREC* '96, November 20–22, Gaithersburg, MD, National Institute of Standards and Technology, Special Publication, pp. 251–263, 1996. (Electronic proceedings at http://www-nlpir.nist.gov/TREC, it can also be found at http://helios.cti.gr/tocs.)

[17] B. Mamalis, P. Spirakis, and B. Tampakas, High Performance Parallel Text Retrieval over Large Scale Document Collections: The PFIRE System, *International Journal of Computers and Their Applications*, to appear.

[18] C. Pogue and P. Willett, Use of Text Signatures for Document Retrieval in a Highly Parallel Environment, *Parallel Computing*, Vol. 4, pp. 259–268, 1987.

[19] G. Salton and C. Buckley, Parallel Text Search Methods, *Communications of the ACM*, Vol. 31, No. 2, pp. 202–215, 1988.

[20] G. Salton and McGill, *An Introduction to Modern Information Retrieval*, McGraw-Hill, New York, 1983.

[21] G. Salton, A. Wong, and C. S. Yang, A Vector Space Model for Automatic Indexing, *Communications of the ACM*, Vol. 18, No. 11, pp. 613–620, 1975.

[22] R. Sharma, A Generic Machine for Parallel Information Retrieval, in *Information Processing and Management*, Pergamon Press, Oxford, pp. 223–235, 1989.
[23] C. Stanfill, Partitioned Posting Files: A Parallel Inverted File Structure for Information Retrieval, *Proc. ACM SIGIR* '90, pp. 413–428, June 1990.
[24] C. Stanfill and B. Kahle, Parallel Free Text Search on the Connection Machine System, *Communications of the ACM*, Vol. 29, No. 12, pp. 1229–1239, 1986.
[25] C. Stanfill, R. Thau, and D. Waltz, A Parallel Indexed Algorithm for Information Retrieval, *Proc. ACM SIGIR* '89, pp. 88–97, June 1989.
[26] H. Stone, Parallel Querying of Large Databases: A Case Study, *Computer*, pp. 11–21, October 1987.
[27] B. Tampakas, K. Antonis, B. Mamalis, B. Papakostas, P. Spirakis, and A. Stamoulis, Efficient Distributed Information Retrieval Techniques with the Vector Space Model, *Proc. ISCA PDCS* '96 *Conference*, September 25–27, Dijon, pp. 726–731, 1996.
[28] F. Tiedt, Parsytec GCel Supercomputer, Technical Report, Parsytec GmbH, July 1992.
[29] C. J. van Rijbergen, *Information Retrieval*, 2nd edition, Butterworths, London, 1979.

Received May 1997, *and in final form April* 21, 1999.