Parallel Techniques For Efficient Searching Over Very Large Text Collections

B. Mamalis P. Spirakis B. Tampakas

Computer Technology Institute
Kolokotroni 3, 26221, Patras, Greece
Tel.: + 3061 220112, Fax.: + 3061 993973
E-mail: {mamalis, spirakis, tampakas}@cti.gr

Abstract

This paper mainly discusses the efficiency of PFIRE system, a parallel VSM-based text retrieval system running on the GCel3/512 Parsytec machine, as well as the effectiveness of the corresponding pre-existing serial FIRE system. Concerning PFIRE, the use of suitable data sharing and load balancing techniques in combination with specific pipelining techniques and with the capability of building binary and fat-tree virtual topologies over the 2-D mesh physical interconnection network of the parallel machine, leads to very fast interactive searching over the large scale TREC collections. Analytical and experimental evidence is presented to demonstrate the efficiency of our techniques. The corresponding conventional FIRE system was also used to measure the effectiveness (in terms of recall and precission) of several IR techniques (statistical phrase indexing, automatic statistical global thesaurus construction etc.) used over the TREC WSJ subcollection.

1. Introduction

Our work consists of two main different parts:

(a) testing (in terms of recall and precision measures) of the previously existing (and appropriately modified for large scale data management) FIRE system over the TREC WSJ subcollection.

(b) integration of the parallel extension of FIRE system (PFIRE) via new advanced parallel techniques (data sharing and balancing techniques, pipelining, fat-trees processing) and extraction of various efficiency results (speed-up, utilization, response times) over the WSJ subcollection and some multiples of it.

Due to the fact that the conventional FIRE system uses rather traditional IR techniques, without something novel to be pointed out (except from our automatic global thesaurus construction approach), we present it (as well as the corresponding TREC results' analysis) quite briefly. On the contrary, we give emphasis (and we describe in detail) on the new techniques (as well as some old ones out of [1]) used for the efficient parallelization of FIRE over the GCel3/512 Parsytec machine. The application of these techniques (which can be strongly recommended for the parallelization of any text retrieval system using the Vector Space Model) over the TREC WSJ collection has led to very satisfactory efficiency results which are suitably analysed and presented (in tables and curves) at the end-sectcion of the paper.

In the following two sections we give a brief description of the basic IR features of FIRE (section 2) as well as a brief presentation of the TREC experiments performed over the WSJ subcollection (section 3). The description of the PFIRE system and a short reference to the hardware platform used (the GCel3/512 Parsytec machine) are given in section 4. Finally in section 5 (which is the most important part of our work), we present the efficiency results of PFIRE over the WSJ collection, as well as the fat-tree processing experiments that were performed with use of the techniques described in [20].

2. The FIRE System

FIRE (Full Information Retrieval Engine, [2]), is a text retrieval system aiming at effective (in terms of recall and precision measures) management and retrieval over large scale document collections. It is based on the Vector Space Model (VSM, [4],[6],[14]) using several text features (terms, phrases, thesaurus classes) as document identifiers. In the following paragraphs we present the basic characteristics of FIRE system (giving emphasis to those used for TREC experiments).

2.1 The Indexing Process

The initial objective of FIRE's indexing processes is to extract the most significant words of each document, and use them as index-terms with appropriate statistically evaluated weights for the corresponding documents.

Both the stopping and the stemming process are applied to all the words of every document. Stopping implies the rejection of the words that are very common in English language. This is done by using a specific stop-list ([7]) that contains the most common words of the language. Stemming implies the extraction of each word's stem, leading to the substitution of the whole set of words in the document collection by the, substancially smaller in size, set of word-stems in the collection. This is done by using the Porter's algorithm as presented in [7]. The resulting word-stems are statistically weighted, (separately for each document), by the inverse document frequency (IDF) method ([6]) which is based on both the frequency of each term in a specific document and the frequency of each term in the whole set of documents.

Finally, a categorization of all the terms according to their document frequencies (DF) is performed. As stated in [14] the most appropriate terms for indexing are those with document frequency $N/100 < DF < N/10$ where N is the number of documents in the collection. The terms with $DF > N/10$ (high DF terms) are appropriate for construction of phrases, and the terms with $DF < N/100$ (low DF terms) are the most appropriate for thesaurus classes' construction. In FIRE, the low DF terms are used both for term-indexing and thesaurus classes' construction and the high DF terms are used for the extraction of phrases. However, during TREC experiments we have used several varying corresponding threshold values (instead of '10' and '100') in order to gain the best possible behaviour of the whole statistical indexing scheme.

According to the above indexing process, each document is finally represented by a corresponding document-vector (following the Vector Space Model) containing three types (single-terms, phrases, thesaurus classes) of weighted identifiers. These document-vectors are then organized into corresponding inverted indicies (separately for each type of identifier).

2.2 Phrase Extraction

The statistical phrase extraction method used in FIRE is similar to the one described in [5]. Following the corresponding general statistical phrase extraction model of the above reference one has to determine three basic parameters: the length of the phrase, the domain in which the searching is performed, and the maximum distance (proximity) of the phrase elements within the specified domain. In FIRE we have used the typical minimum values for the above parameters (Length = 2, Domain = sentence, Proximity = 1) leading to the extraction of adjacent pairs of words within a sentence.

Following the considerations of paragraph 2.1 we have mainly used the high-DF terms of the document collection as phrases' terms. Specifically (as it is defined in [5]), at least one term of the phrase must be a high-DF term (based on parameter DFh), whereas the document-frequency of the whole phrase itself must exceed a specific threshold (parameter DFp). The resulting phrases (and the corresponding weights) naturally vary according to the above threshold values.

2.3 Automatic Thesaurus Construction

The automatic construction of the thesaurus classes in FIRE, is performed using the following procedure which consists of three main steps:

(a) Documents' clustering

First, the similarities of all pairs of documents in the collection are computed forming the corresponding similarity matrix. This computation is performed by the application of the well-known cosine similarity function ([6]) on the two term-vectors of each document-pair.

Given the above similarity matrix and one similarity threshold, several clustering algorithms based on graph theory can be used in order to extract corresponding documents' clusters. In FIRE, an algorithm based on the iterative evaluation of Connected Components (CCs) of controlled size, is used. This algorithm is briefly described below:

<u>Assume that</u>:
D : the whole set of the N documents in the collection
MINTH : the desirable min number of docs per cluster
MAXTH : the desirable max number of docs per cluster
STEP : the increament value for similarity threshold
INTH : the initial similarity threshold
STEPTH : the current similarity threshold

<u>The algorithm:</u>

```
INTH = 0 ;
STEPTH = INTH + STEP ;
while (D NOT emptyset)
  begin
    Compute the CCs of D using STEPTH as simi-
         -larity threshold for each CC
    for each CC do
      begin
        Let n be the # of docs in the CC
        if (n < MINTH)
          Ignore CC and remove its documents from D
        else if (n > MAXTH)
          Ignore CC (set D remains unchanged)
        else
          Accept CC as a proper document cluster
```

end
Increase STEPTH by the value STEP
end

The above algorithm finally leads to document clusters of size (# of documents) between MINTH and MAXTH. Although the Connected Components algorithm may be considered much simpler compared to existing algorithms (e.g. see the complete-link algorithm in [8],[9]), it is quite efficient when it is coupled with the appropriate parameter values.

(b) Extraction of the thesaurus classes

When the document clusters have been determined, the thesaurus classes themselves can be generated (one class for each cluster). The simplest and most reliable approach is to define a thesaurus class as the intersection of all the low frequency terms of the documents in a cluster. Alternate methods refer to the use of other set-operations such as union and restricted intersection. In FIRE an approach of extended intersection is used (extended, stands for the fact that this approach leads to classes with more terms than the simple intersection). It is supported by a user supplied parameter that determines the percentage of documents in a cluster that must contain one low frequency term in order for this term to be included in the corresponding thesaurus class.

(c) Expansion of the document term vectors by thesaurus classes

Once the thesaurus classes are generated, all documents in the collection can be indexed by "replacement" or "augmentation" ([8],[9]). Replacement entails replacing a term in a vector by a thesaurus class which contains that term. Augmentation involves adding the thesaurus class to the vector; the original term remains in the vector unchanged. We believe that augmentation is more effective than replacement. Another interesting point during the expansion of the term-vectors by thesaurus classes is the weighting of these thesaurus classes. Based on corresponding experiments, we conclude that the weighting function described in [8],[9] is quite effective (that weighting fuction has been used in FIRE with varying parameter values for down-weighting the resulted classes for each document).

2.4 The Retrieval Process

The documents' indexing process described in 2.1 is also used for indexing the queries. The query terms are filtered through the stop-list test and the stemming process, and then they are incorporated into the query term-vector (weighted by the standard value of one). The query term-vector has the form of the document term-vectors. The expansion of the query term-vector by the appropriate phrases (extracted via the same method described in 2.2) and thesaurus classes (that contain at least one term of the query) directly follows.

The next step is to compare the query vector (following the Vector Space Model) to all the document vectors that represent the documents of the collection. The above comparison is based on the evaluation of the cosine-similarity value between the query and each document in the same way that the document to document similarity is evaluated in 2.3. The difference here is that the phrases and the thesaurus classes that have expanded both the query and the documents also take part in the query-document similarity computation. The final step is the ranking of the documents according to their scores (regarding the comparison to the query). The documents with the highest scores are then considered as the most relevant ones to the query.

2.5 Other Features

Two other significant features of FIRE system are the relevance feedback and passage retrieval methods used. The Ide dec-hi formula (see [12]) has been used for interactive relevance re-formulation of the queries, whereas the passage indexing and retrieval method used in FIRE is similar to those described in [10],[11]. The above characteristics of FIRE will not be described in more details since they have not been used for TREC experiments.

Also, FIRE is supported by a well-designed, windows-based (X-Windows, OSF/Motif), user interface. The user interface is generally seen as a key component to successful development, affecting important factors like the long-term durability of a software product. We studied the user interface of several known text retrieval systems like the SMART ([15]), the BASIS and the ARCHEION systems (the last two systems are commercial information retrieval products) and we have incorporated their most important key fatures in our implementation.

Another feature is the capability of defining categories over the whole document collection. Consequently, users are capable of selecting specific categories that will then participate in the retrieval process. Thus, the users are capable of directing the whole retrieval process closer to their needs. Finally, suitable distributed IR techniques have recently applied on FIRE system, providing the capability of efficient operation over highly distributed environments (see [3]).

3. TREC Experiments over FIRE

The retrieval effectiveness (in terms of recall and precision measures) TREC experiments that have been performed over FIRE system aimed at mainly testing the basic IR features of FIRE over large document collections. In the following we briefly present these experiments and a draft analysis of the corresponding results.

3.1 Description of Experiments

The experiments have been performed over the:

- Ad-hoc topics (251-300, Description fields only)
- Category B data that consist of the WSJ subcollection (about 75000 documents, 250 Mbytes)

We've performed two official Ad-hoc automatic experiments (Ctifr1 and Ctifr2 runs) in which the following features of FIRE system were used for both the documents and the queries (the relevance feedback and passage retrieval methods have not been used yet for TREC data):

- stopping, stemming
- statistical word indexing
- statistical phrase extraction
- automatic global thesaurus construction

In both the experiments we tried to use suitable combinations of all the parameter values that had to be specified (high-DF and low-DF thresholds for phrase and thesaurus construction, special parameters for thesaurus construction – MINTH, MAXTH – etc.) The following parameter values were used for each run:

Ctifr1	Ctifr2
High-DF threshold: $\frac{N}{10}$	High-DF threshold: $\frac{N}{4}$
Low-DF threshold: $\frac{N}{1000}$	Low-DF threshold: $\frac{N}{500}$
MINTH: 20	MINTH: 10
MAXTH: 50	MAXTH: 30

(Ctifr2 parameter values lead to fewer phrases and smaller and more tight thesaurus classes than Ctifr1 run)

3.2 Statistics of the experiments

The serial experiments were performed on a (shared) SUN/Sparc Station 10 with 110 MHz, 32 MBytes main memory and 2 GBytes disk storage.

The total size of the inverted indicies built for all the types of identifiers (terms, phrases, thesaurus) was 220 MBytes for Ctifr1 and 180 MBytes for Ctifr2.

Time spent for the preprocessing phase:

- 5.5 hours for Ctifr1 (2.5 hours for terms and phrases, 3 hours for theasurus construction
- 4.5 hours for Ctifr2 (2 hours for terms and phrases, 2.5 hours for thesaurus construction

Average searching time per query : 60 seconds.

3.3 Analysis of Results

Considering the official testing of the conventional FIRE system over the WSJ collection the following remarks should be pointed out:

- Generally, the official recall/precision based results for both runs were not very good (specially concerning the average precision values over all the description topics), although that FIRE system has been tested succesfully (with quite good recall/precision measures) over the old, very small in size, standard collections (Cranfield and Time).
- The results for Ctifr2 (fewer phrases and small and tight thesaurus classes seem to be a little better than the Ctifr1 results.
- There were some "very bad" topics that affected significantly the recall capability of the system (namely the ad-hoc topics 251, 264, 269, 270, 273, 274, 285, 291, 294 which need either tokenizer and syntactic analysis or thesaurus-of-synonyms support).

  Specifically, in a total of 45 topics that had at least one relevant document we had (in Ctifr1 run): 1064 relevant docs in the collection – 378 relevant docs retrieved that is divided to:

  1) 474 rel docs totally – 296 rel docs retrieved, for the 36 good queries and
  2) 590 rel docs totally – 82 rel docs retrieved, for the 9 "very bad" queries

- However, the above bad queries did not affected so significantly the average precision values.

Trying to explain the relative retrieval failure of our experiments we could say that (beyond the large size of the collection and the advanced philoshophy of the corresponding topics).

(a) the relevance feedback and passage retrieval methods of FIRE system were not used (due to the lack of time)

(b) there are several significant features (especially for TREC runs) that our system misses such as:

   i. Syntactic documents' parsing and syntactic phrase indexing
   ii. Tokenizer for dates, phone numbers etc. in documents
   iii. Externally built auxiliary files (thesaurus)
   iv. Advanced automatic query extraction, formulation and re–formulation methods (that seems to be the most important for TREC experiments)

4. The Parallel Extension of FIRE System (PFIRE)

The parallel extension of FIRE system has been based on a quite efficient VSM-based parallel binary-tree algorithm ([1]) which has been appropriately implemented on the GCel3/512 massively parallel machine. Generally, a lot of parallel text retrieval attempts have been done in the past with use of various hardware platforms and text processing models (e.g. [17],[18],[19]. The developement and implementation of PFIRE resulted to a quite friendly and very efficient multiprocessor text retrieval system, preserving all the IR features of the conventional FIRE system as well as the X-Windows graphical user interface of FIRE. In the following paragraphs we briefly present the basic components of PFIRE (the hardware platform and the parallel algorithm).

4.1 The GCel3/512 Parsytec Machine

The hardware that we use in our implementation is the Parsytec GCel3/512 machine ([13]) which belongs to the MIMD class of parallel computers. It is a massively parallel machine consisted of 512 processor-units formed in a two dimensional grid interconnection network with dimensions 32 x 16. Each processor-unit is an Inmos T805 transputer, a member of the well known transputer family, which has the following characteristics:

- 32 bit RISC processor with frequency 30 MHz and peak performance 4.3 MFLOPS, 30 MIPS
- 4 KBytes on–chip cache memory and 4 MBytes external DRAM local memory
- 4 bidirectional high speed links of max data rate: 20 Mbits/sec (unidirectional) and 18.8 Mbits/sec (bidirectional)

The 512 transputer processors of the GCel3/512 computer are organized in a hierarchical structure. Each transputer is a processing unit. Eight processing units form a processing board. Two processing boards compose a cluster. A cluster (containing 16 transputers) is the atom of the GCel3/512 machine and it's the smallest unit that can be accessed by a user. Four clusters make a GigaCube and eight gigacubes make the whole GCel3/512 machine. From the user's point of view the whole 512-transputer network is split up in partitions of varying sizes. Each partition consists of one or more clusters. A partition may contain parts of other partitions.

The two-dimensional grid network of transputers is connected to the outside world through 2 external hosts which provide 12 high speed links (20 Mbits/sec). The hosts are two Sun/SPARC stations running the PARIX operating system. PARIX (PARallel extensions to unIX) is a UNIX–based operating system with extensions (tools and programs) that support the transputers' communications with each other. PARIX is responsible for the efficient usage of the transputers' grid network.

4.2 The VSM-Based Parallel Algorithm

The main objective of the VSM-based parallel algorithm ([1]) used in PFIRE is to appropriately minimize the total communication times needed for the whole execution. Towards this direction, we've developed a virtual binary-tree processors' organization scheme over the 2-D mesh structured interconnection network of the machine. A brief description of the algorithm directly follows:

We suppose that we are given a complete binary-tree network of P processors where all of them ((P+1)/2 leaves and (P-1)/2 internal nodes) serve as working processors. Also we are given a VSM-indexed text query (query vector q_i) and a VSM-indexed text collection of D documents (and D document vectors as a consequence) which is appropriately shared over the local memories of the P processors (let's say approximately D/P vectors for each processor). Also there is a host processor which connects the root of the tree to the outside world and initially holds the query vector q_i. Under the above settings, the binary-tree single-query algorithm briefly consists of the following steps:

1. The query vector q_i is sent to all the processors of the tree progressively. Beginning from the root, each processor sends it to its two children and so on, till the query vector arrives at the leaves after logP steps (plus 1 step for the host-to-root communication).
2. Each processor performs the scoring and ranking subtasks over its own approximately D/P document vectors, thus resulting to one ranked relevant documents' set (RD) of size R for each processor.
3. The local ranked RD sets are merged progressively through all the levels of the tree (in parallel for the processors of the same level). Beginning from the lowest level (the parents of the leaves) each processor merges the RD sets of its 2 children into his own RD set, sends the result (of size R) to its parent and so on, till one final RD set is extracted by the root processor after logP such merging steps. This final set is then sent to the host processor (1 more step).

As it comes out of [1],[20], the total time spent for the execution of the above algorithm can be given by the following formula:

$$T = (2logP + 2) \; + \; (T_s + T_{lr}) \; + \; 4R(logP - 1) \quad (1)$$

where (a) $(2logP+2)$ is the total number of communication messages that are exchanged during the execution of all the phases of the algorithm ($logP + 1$ from step

1 and $logP+1$ from step 3), (b) (T_s+T_{lr}) is the total computational time needed and it is almost equal to $\frac{1}{P}(t_s+t_{lr})$ – t_s, t_{lr} represent the serial times for scoring and ranking respectively, and (c) $4R(logP-1)$ is the merging time ($4R$ for merging of 3 RD sets in one node and $logP-1$ because this task takes place sequentially in all the $logP-1$ levels of the tree).

Concerning the performance of the above algorithm, we can observe that the communication times remain logarithmic, whereas the computational tasks are parallelized almost perfectly (see [1] for further explanation). The corresponding experimental results (presented in [1]) gained by the application of the algorithm over the GCel machine (via the use of virtual tree topologies over the physical 2D-mesh network of the machine) validate a very efficient behaviour over the, small in size, CRANFIELD and TIME text collections and over simulated large scale text collections (formed by reproducing the CRANFIELD collection multiple times). In the next section we demonstrate the outstanding efficiency of the above algorithm (extended by data balancing and pipelining methods) even when used over the large scale TREC WSJ subcollection.

5. TREC Experiments over PFIRE

5.1 Description of Experiments

Various efficiency TREC experiments have been performed over PFIRE towards the direction of gaining very fast interactive response times over a large scale "real" text collection. The previously (4.2) mentioned binary-tree algorithm (extended by new parallel techniques described in 5.2) was used, as well as our new multiple-query fat-tree algorithm ([20]). All the experiments have been performed over the GCel3/512 Parsytec Supercomputer (4.1). The collection data used were

- The TREC WSJ subcollection (about 250 MBytes)
- Parts (1/2 and 1/4 WSJ) and simulated multiples (up to 8–times WSJ) of the WSJ subcollection (by reproducing the WSJ appropriately) which result up to 2 GBytes of indexed–data

whereas the Ad-hoc TREC topics (251-300) have been used as user queries in order to obtain the desirable average performance measurements per query for all the experiments.

5.2 Modifications on the Binary-tree Parallel Algorithm

As mentioned above (and clearly pointed out in [1]) the existing binary-tree parallel algorithm gives very good results (up to 80% utilization and very fast response times) in the general case and was expected to give (via simulations) excellent results when the data are uniformly distributed over the processors of the machine (up to 90% utilization). Trying to gain even better results (for both response times and speed-up and utilization measures) over "real" large collection data we've performed the following challenging modifications:

(a) Data sharing and balancing

Obviously (with regard to the binary-tree algorithm of [1]), the way that the D document vectors of the whole collection are shared to the P processors of the machine is quite critical because it directly affects the distribution of the computational load (which consists of the local scoring and local ranking tasks) over the P processors. Moreover, it's obvious that if the computational load of even one processor is much larger than the computational load of the other processors, it could result to significant undesirable increase of the total computational times (T_s+T_{lr}) due to the fact that the merging phase of the algorithm requires the completion of the local scoring and ranking tasks of all processors, in order to be correctly terminated.

Generally, considering the above problem (and having in mind that the computational load of each processor consists of the local scoring and ranking tasks) we can state the following basic requirements concerning the desirable data-sharing and balancing scheme of the D document vectors over the P processors of the machine:

- From the *"local ranking"* point of view, the most preferable solution requires each processor to pocess the same number of document vectors (let's say approximately $\frac{D}{P}$ vectors).
- From the *"local scoring"* point of view, the most preferable solution requires each processor to pocess the same amount of data in it's own local memory (the sum of the sizes of it's own vectors), independently to the number of these vectors).

Obviously, it's quite easy to satisfy the first requirement by randomly loading $\frac{D}{P}$ vectors to each processor (e.g. the first $\frac{D}{P}$ vectors to the first processor etc.). However, the satisfaction of the second requirement implies the use of a time-consuming sorting-based algorithm. Moreover, it is very hard for any existing algorithm to satisfy optimally both the above requirements considering the usual case of a text collection with varying, in size, document vectors (there is an obvious contradiction between the two requirements – the achievement of equal amount of data on each processor normally leads to different number of vectors for each processor).

In the initial implementation of PFIRE we had followed the easy solution of satisfying the first requirement only, having the advantage of the direct (not time-consuming) loading of the processors of the machine, and the disadvantage of undesirable computational overheads due to the varying sizes of the document vectors of the VSM-indexed text collection. On the contrary, during TREC experiments we tried to develop a data-sharing and balancing scheme that satisfies both the above requirements (as much as possible) towards the direction of gaining faster response times and even better speed-up and utilization measures.

Our data-sharing scheme is naturally based on the initial sorting of all the document vectors according to their sizes. Afterwards, it tries to distribute the sorted document vectors over the P processors in such a way that each processor finally pocess the same number of vectors and almost equal amount of data, providing that the variations on the sizes of the different vectors are of normal extent. The key idea of this specific distribution is to serially divide the sorted set of vectors on subsets of size P, and then share each subset (by one-to one mapping) to the P processors in such a way that each time the corresponding subset is mapped to a different partition of the processors' set.

More specifically, given a VSM-indexed (with any number of different types of identifiers – words, phrases and thesaurus classes in our case) text collection of D documents (that leads to D corresponding document vectors) and P processors as an ordered set $P_{set} = (1, ..., P)$, our off-line data balancing algorithm can be described (in a more formal way) as follows:

A. Sort the D document vectors according to
their size, thus leading to an ordered set
$D_{sort} = (D_1, ..., D_i, ..., D_D)$,
where D_1 is the vector of the maximum
size and D_D denotes the vector of the
minimum size.
B. *for* $i = 0$ *to* $(\lfloor \frac{D}{P} \rfloor - 1)$
   1: Distribute the $(D_{1+(i \times P)}, ..., D_{(i+1) \times P})$
      vectors of D_{sort} to the processors of P_{set}
      by one-to-one mapping (e.g. $(D_{1+(i \times P)}$
      is given to the first processor of P_{set} etc.
   2: Shift P_{set} one position right (e.g. with
      one shift P_{set} becomes $(2, ..., P, 1)$)
C. Distribute the remaining $(D_{1+(\lfloor D/P \rfloor \times P)}, ..., D_D)$
vectors of D_{sort} to the first $D - (\lfloor D/P \rfloor \times P)$
processors of the last shifted P_{set}

Obviously, the above algorithm leads to equal number of vectors for each processor. Beyond this, with regard to the requirement of giving almost the same amount of data to each processor, it's clear that the effectiveness of our scheme is collection dependent. However, it clearly promises much better results (concerning the respose times, speed-up and utilization measures) comparing to the method that was initially used in PFIRE. Now, considering the TREC collections, we can say that the corresponding variations on the sizes of the document vectors are quite extended, however they are not extreme. Moreover (as it is clearly validated by the experimental results presented in section 5.3), the improvements offered by the use of our data-sharing method are quite significant and the undesirable computational overheads caused by the initially used random data distribution are properly eliminated.

(b) Pipelining

The pipelining techniques that have been incorporated into the binary-tree algorithm aim at the appropriate minimizing of the average query waiting times when multiple queries are to be processed together (the one after the other) by the P processors of the machine. Specifically, supposing that we are given X user queries that are to be treated together by our binary-tree algorithm (the one after the other) we introduce the following pipelined steps:

1. *During the construction and the propagation of the user query through the tree.* Specifically, the host processor is allowed to construct and send sequentially all the queries that have arrived to him, to all the processors independently to the completion of the remaining tasks for each query. It's obvious that the total time needed for the execution of the above pipelined task for the propagation of all the X queries is equal to $logP + X$ message steps ($logP/X + 1$ in average for each query), whereas without pipelining we had to spend $logP + 1$ message steps for the propagation of one query only.
2. *During the merging phase.* Specifically, each processor, after the completion of the local scoring and ranking tasks for one query, is allowed to continue by performing local scoring and ranking for the next query, independently to the completion of the merging task of the first query. Obviously, one processor can behave in such a way, either in the case that he waits for the RD sets of his two children or in the case that he has sent his RD set (merged with the RD sets of his two children) to his parent. Moreover, the above pipelining can be reliably be applied on the binary-tree algorithm since each processor receives all the X queries from the beginning (this is achieved via the previously mentioned pipelining on the construction and propagation of the user queries). We also have to note here that the implementation of such a pipelining technique requires very careful treatment because of the complicated synchronization that is introduced.

Now, considering the average waiting time improvements achieved per query with use of the above pipelining techniques (with regard to equation (1) and the corresponding discussion of section 4.2) we have to outline the following remarks:

- The logarithmic factor of the query propagation message steps is almost eliminated (we refer to the $logP+1$ query message steps out of the $(2logP+2)$ totally needed communication steps).
- The merging overhead (initially caused due to the need of the sequential execution of the merging phase through the $logP$ levels of the tree) is reduced quite significantly. Actually, the logarithmic factor of the merging computational task (we refer to the factor $4R(logP-1)$ of equation (1)) is almost eliminated.

Moreover, considering the large size of the RD sets required in TREC experiments (equal to '1000', thus resulting to increased merging overhead), we can say that the corresponding improvements are expected to be quite clear. However, we have to note here that the logarithmic factor of the RD message steps (the half of the $(2logP+2)$ totally needed communication steps) can not be eliminated (which means that the logarithmic factor concerning the total communication times still remains). Actually, the complete elimination of the logarithmic factors of equation (1) (that implies the substantial minimization of the communication times of our algorithm) requires the use of more complex pipelined parallel algorithms running on higher capacity interconnection networks (than a simple binary–tree topology and the 2-D mesh structure of the GCel3/512 machine). For example, see [20] where a *fat–tree* is appropriately used or section 5.4 for a brief description.

The corresponding experimental results (presented in the following section) demonstrate the quite efficient behaviour of the above pipelining techniques and they validate all the above theoretically stated considerations.

5.3 Efficiency of Searching

In order to obtain a realistic evidence concerning the high performance of PFIRE system over the large scale TREC WSJ collection (both with and without the use of the data–sharing and pipelining techniques) we've performed various efficiency experiments (under the settings of section 5.1) for the following separate cases:

- with and without data sharing and balancing
- varying collection size
- varying number of retrieved documents (R)
- with and without pipelining

In all the above cases the corresponding measurements have been taken for varying number of processors (ranging from 16 to 512). The performance measures evaluated in each case are the well–known *speed–up* (S_P) and *utilization* (U_P) measures which can be defined as

$$S_P = \frac{T_1}{T_P} \;,\; U_P = \frac{S_P}{P}$$

where T_1 is the time to carry out the whole system's work on one processor and T_P corresponds to the time needed when P processors are used for the same work in parallel. In the ideal case, $S_P = P$, whereas the utilization measure denotes (practically) the fraction of the ideal speed–up that is achieved in each case. Also, the response times (R_P, in seconds) of the system are measured for all the experiments.

(i) Data sharing and Balancing

Concerning the worth of using the data sharing and balancing scheme described in 5.2, we present in tables 1 and 2 extended comparative experimental results gained by the execution of the binary-tree algorithm both with (table 2) and without (table 1) data–balancing. Specifically, in both the tables the response times and the speed–up and utilization measures are given for varying number of processors (from 16 to 512) and for varying collection size. With regard to the collection size we have used the WSJ collection, a fraction of the WSJ of almost 60MB (1/4–WSJ) and a simulated multiple of the WSJ of almost 1.9GB (the 8–WSJ). Also, for both the experiments we assume that the value of R (# of retrieved documents) is equal to '1000'.

The first conclusion that can easily be extracted from tables[1] 1,2 is that the results gained by the use of data–balancing (table 2) are substantially better than the ones offered by the initial binary-tree algorithm of PFIRE. As an example, the utilization measure for 64 processors over the 1–WSJ collection increases to 90% (from 78% without balancing) and finally becomes 97% for 512 processors over the 8–WSJ (from 80%). Improvements of similar extent hold for the response times too (e.g. for 512 processors over the 1–WSJ collection we obtain response time 0.8 seconds with data–balancing and 0.99 seconds without balancing).

Moreover, the reader can easily notice that the corresponding improvements sligtly increase *(a)* with the increase of the # of processors (e.g. the difference 90%–to–78% for the 1–WSJ and 64 processors becomes 80%–to–53% for 512 processors) and *(b)* with the increase of the collection size (e.g. the difference 38%–to–22% for 256 processors and the 1/4–WSJ becomes 85%–to–63% for the 1–WSJ). The *former* ((a)) can naturally be explained by the fact that for larger # of processors (and constant collection size which means constant # of vectors) the random sharing of the data–vectors initially used by PFIRE leads to worse distribution since each processor gets fewer document vectors.

[1] The '–' in some cells means that the corresponding x–WSJ collection does not fit in the total memory of the P processors.

P	Speed-up			Utilization			Response Time		
	1/4 WSJ	1 WSJ	8 WSJ	1/4 WSJ	1 WSJ	8 WSJ	1/4 WSJ	1 WSJ	8 WSJ
16	11.2	--	--	70%	--	--	5.25	--	--
32	19.5	--	--	61%	--	--	3.12	--	--
64	32	49.9	--	50%	78%	--	1.78	2.51	--
128	46.1	89.6	--	36%	70%	--	1.11	1.79	--
256	56.3	161.2	--	22%	63%	--	0.89	1.32	--
512	54.8	271.3	408.8	14%	53%	80%	0.78	0.99	3.31

Table 1: Measurements for varying collection size without balancing

P	Speed-up			Utilization			Response Time		
	1/4 WSJ	1 WSJ	8 WSJ	1/4 WSJ	1 WSJ	8 WSJ	1/4 WSJ	1 WSJ	8 WSJ
16	12.9	--	--	81%	--	--	5.12	--	--
32	23	--	--	72%	--	--	2.87	--	--
64	39.6	57.6	--	62%	90%	--	1.61	2.39	--
128	61.4	112.6	--	48%	88%	--	0.97	1.62	--
256	97.2	217.6	--	38%	85%	--	0.68	1.13	--
512	158.7	409.6	496.6	31%	80%	97%	0.52	0.8	3.05

Table 2: Measurements for varying collection size with balancing

P	Speed-up			Utilization			Response Time		
	50	100	1000	50	100	1000	50	100	1000
64	59.2	58.7	57.6	92.5%	91.8%	90%	2.08	2.19	2.39
128	114.8	113.9	112.6	89,7%	89%	88%	1.37	1.44	1.62
256	224,2	222,7	217.6	87.6%	87%	85%	0.98	1.02	1.13
512	424,9	420,3	409.6	83%	82.1%	80%	0.7	0.73	0.8

Table 3: Measurements for varying # of retrieved documents with balancing

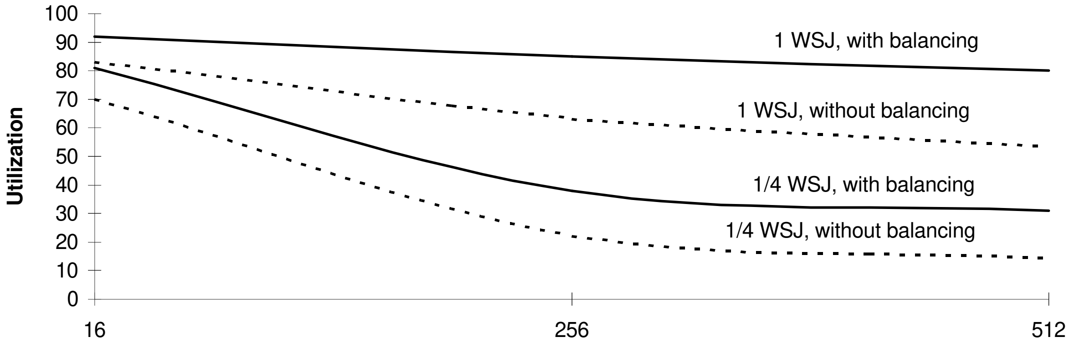

Figure 1: Utilization curve for varying # of processors with and without balancing

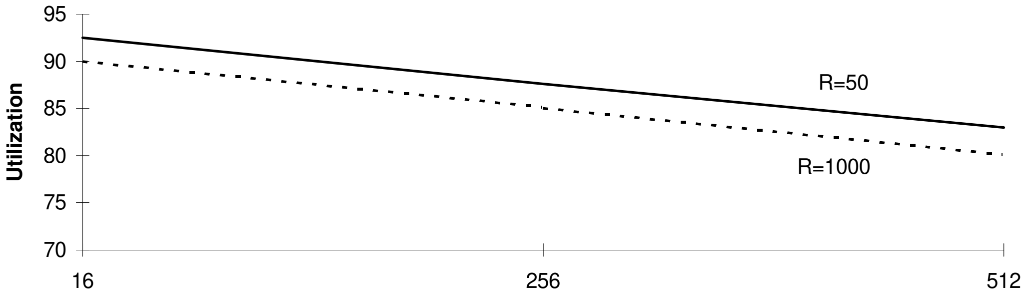

Figure 2: Utilization curve for varying # of processors and retrieved docs

P	Speed-up			Utilization			Response Time		
	1/4 WSJ	1 WSJ	8 WSJ	1/4 WSJ	1 WSJ	8 WSJ	1/4 WSJ	1 WSJ	8 WSJ
16	13.9	--	--	87%	--	--	4.83	--	--
32	25.2	--	--	79%	--	--	2.7	--	--
64	44.8	59.5	--	70%	93%	--	1.5	2.11	--
128	78	117.7	--	61%	92%	--	0.89	1.49	--
256	135.6	230.4	--	53%	90%	--	0.63	1.06	--
512	250.8	445.4	501.7	49%	87%	98%	0.49	0.75	3.01

Table 4: Measurements for varying collection size with pipelining

P	Speed-up		Utilization	
	with	without	with	without
16	12.9	11.2	81%	70%
32	27.2	24	85%	75%
64	57.6	49.9	90%	78%
128	119	101.1	93%	79%
256	245.7	203.5	96%	79.5%
512	501.7	408.8	97%	80%

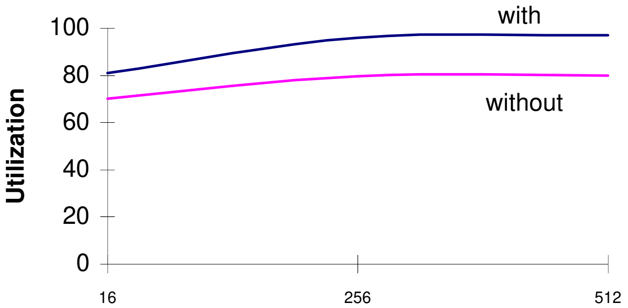

Table 5: Measurements for full-of-data processors with and without balancing

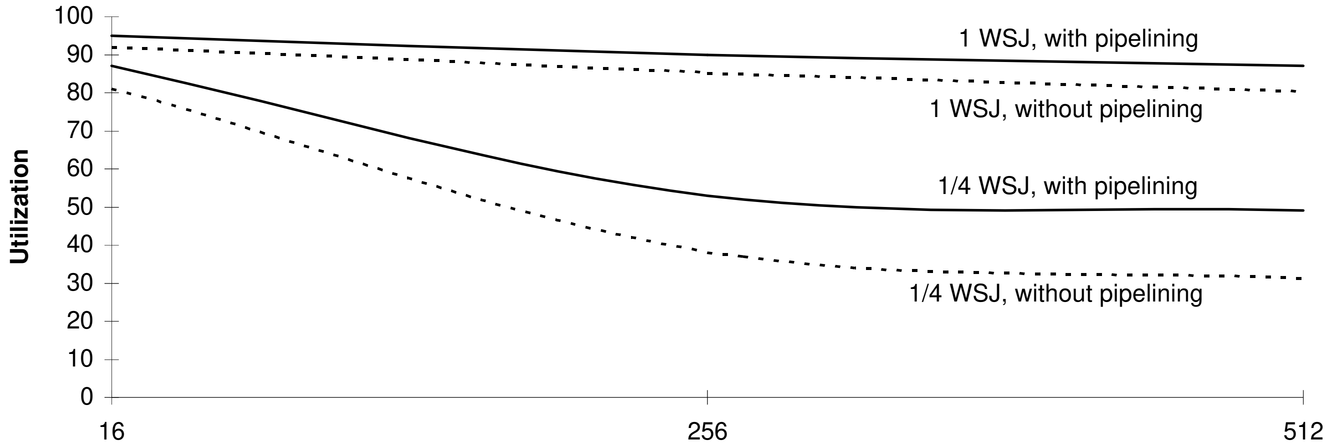

Figure 3: Utilization curve for varying # of processors with and without pipelining

Thus, the corresponding overhead grows up and the difference comparing to a suitably balanced indexed-data set tends to be substantial. On the other hand, concerning the *latter* ((b)) one would expect the opposite effect since the use of a larger collection implies more document vectors for each processor and the result of the random data–vectors sharing is expected to be quite better. However, when larger collections are used (as it is clearly explained in the directly following paragraphs) the positive effect due to the inrease of the computational load for all the processors is much more critical and finally it dominates over any other factor. A quite clear view of the above conclusions is given also by the corresponding utilization curves presented in figure 1 (two pairs of curves, one for the 1/4 WSJ and one for the 1–WSJ collection). Finally, we have to note here (this note holds for all the experiments presented in section 5) that although the achieved response times are very good, much better response times could be achieved if we had assumed lower values of R. Such experiments (for $R = 10$) which lead to excessively low response times for large # of processors can be found in [20].

(ii) Varying collection size

Based on the experimental results presented in both the tables 1 and 2 we can also extract some quite interesting conclusions concerning the influence of the varying collection size on the efficiency of our algorithm, independently to the use of the data sharing and balancing scheme. First, it's obvious that, for constant number of processors, the speed–up and utilization measures increase significantly with the increase of the collection size (e.g. for 128 processors and with the use of data–balancing – table 2 – the utilization measure is 48% for the 1/4–WSJ collection and increases to 88% when the 1–WSJ collection is used. This obviously happens because the use of a large collection implies larger amount of data loaded to each processor which results to larger computational times. Consequently, the increased computational times dominate over the communication times (which remain constant for constant # of processors), thus leading to increase of the achieved speed–up and utilization values.

Following the same considerations (as they can stated in a similar way for small, in size, collections) two other interesting observations coming out from tables 1,2 can be naturally explained: (a) all the speed–up and utilization values decrease significantly with the increase of the # of processors when the collection size remains constant and (b) we obtain quite low speed–up and utilization values when the collection size used is too small for the total memory capacity of the P processors (e.g. 31% and 14% utilization when the 512 processors and the 1/4–WSJ collection are used, with and without data–balancing respectively). As a general conclusion we can say that, given a constant number of processors, the best possible speed–up and utilization values would be achieved in the case that all the processors are full of indexed data. Again, the utilization curves presented in figure 1 can serve as a more clear demonstration of all the above conclusions.

Furthermore, in order to obtain additional evidence concerning the grate positive effect of the large collection size over the performance of our algorithm (as well as for the outstanding efficiency of the algorithm itself in that case) we have performed another very interesting experiment (see table 5) which outlines the speed-up and utilization measures for various values of P (with and without data–balancing), providing that in each case the P processors are full of data. The corresponding parts and multiples of the WSJ collection that have been used in order to fill the P processors with indexed-data are the following: 1/4–WSJ for 16 processors, 1/2–WSJ for 32, 1–WSJ for 64, 2–WSJ for 128, 4–WSJ for 256 and 8–WSJ for 512 processors).

As it comes out from table 5, the values for both the speed-up and utilization measures are very high for all the varying values of P. As an example, when data–balancing is used the utilization values start at 81% for 16 processors and raise up to 97% for 512 processors. Also, we can observe that in this case (full of data processors) the utilization values increase with the increase of P (as it is opposed to the case of using the same collection size for all the different numbers of processors – see tables 1,2). The above observation proves that the collection size is the most important factor concerning the performance of our parallel algorithm. Additionally (as it was expected), both the speed-up and utilization are quite better when data–balancing is used. However, the values for the case that data–balancing is not used are also very satisfactory. Finally, in the figure next to table 5, the reader can find the corresponding utilization curves which lead to the same conclusions reffered above. Both the curves start at very high point–values and slightly raises up to even higher values (as opposed to the corresponding curves of figure 1).

(iii) Varying number of retrieved documents

As mentioned above, all the measurements of tables 1 and 2 have been taken for value of R (# of retrieved documents) equal to '1000' (as it is required for TREC experiments). Obviously, this large value of R causes additional computational overheads because it directly affects the total merging time needed for the execution of the algorithm (see equation (1)). Thus (in order to demonstrate the corresponding difference) we present in table 3 some experimental measurements for more "*realistic*" values of R ('50' and '100') and we compare them to the ones (included in table 3 too) for R=1000. All the measurements of table 3 have been performed for the 1–WSJ collection and with use of data–balancing.

As it comes out directly from table 3 all the measured values (speed-up, utilization, response times) become quite better for R=50 or R=100 (comparing to the values for R=1000). Moreover, we can observe that the utilization measure for R=50 and 64 processors (where the 1–WSJ collection fits exactly) raises up to 92,5% (quite significant improvement comparing to the 90% utilization value for R=1000. The corresponding differences can also be observed out of the utilization curves for R=50 and R=1000 that are presented in figure 2. We have to note here, that all the differences (specially concerning the speed-up and utilization measures) would be even more significant if the data–balancing technique was not used. As a general conclusion (considering the results of table 3 and figure 2) we could say that for smaller (and more realistic) numbers of retrieved documents our algorithm (including data–balancing) becomes even more efficient and it leads to very high utilization values even when the processors of the machine are not full of data (e.g. almost 90% for R=50, 128 processors, 1–WSJ collection).

(iv) Pipelining

Finally, in table 4, we present the experimental results gained by allowing pipelining during the merging and the query propagation phases of our algorithm. Generally, table 4 has the same form with the previously examined tables 1,2, whereas all the measurements presented in this table have been taken for value of R equal to '1000' and with use of data–balancing (thus,

P	Speed-up			Utilization			Response Time		
	1/8 WSJ	1/4 WSJ	1 WSJ	1/8 WSJ	1/4 WSJ	1/2 WSJ	1/8 WSJ	1/4 WSJ	1/2 WSJ
8	6.64	--	--	83%	--	--	4.46	--	--
16	12.16	13.92	--	76%	87%	--	2.65	4.83	--
32	22.08	25.28	25.92	69%	79%	81%	1.6	2.7	4.03
64	38.4	44.8	46.72	60%	70%	73%	0.89	1.5	2.71

Table 6: Measurements for the binary-tree algorithm

P	Speed-up			Utilization			Response Time		
	1/8 WSJ	1/4 WSJ	1/2 WSJ	1/8 WSJ	1/4 WSJ	1/2 WSJ	1/8 WSJ	1/4 WSJ	1/2 WSJ
8	6.72	--	--	84%	--	--	4.23	--	--
16	12.64	14.24	--	79%	89%	--	2.35	4.61	--
32	23.68	26.56	27.2	74%	83%	85%	1.39	2.47	3.68
64	42.88	48	50.56	67%	75%	79%	0.73	1.31	2.39

Table 7: Measurements for the fat-tree algorithm

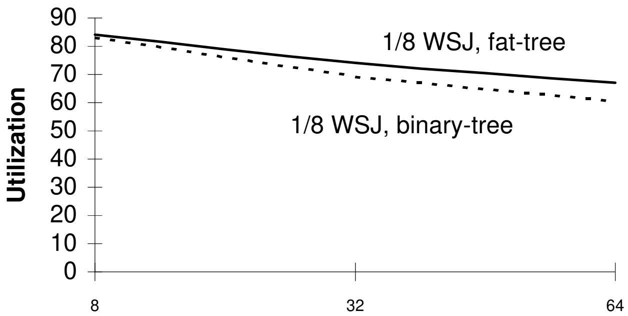

Figure 4: Utilization curves

relative comparisons should be done with table 2).

More specifically, by comparing tables 2 and 4, we observe an obvious improvement (concerning all the performance measures – speed–up, utilization, response times) offered by the use of our pipelining techniques (table 4). Moreover, the reader can notice that this improvement slightly increase with the increase of the number of processors (P) of the machine. This happens because the corresponding time–saving (the logarithmic factors, as it is explained in 5.2) is affected almost directly by the logarithm of P. The above conclusion can be extracted more clearly by observing the corresponding utilization curves in figure 3 (two pairs of curves, for the 1–WSJ and the 1/4–WSJ collection respectively). Additionally, as it comes out from figure 3 the improvements offered by the pipelining are larger when the smaller collection is used (the 1/4–WSJ collection). This happens because for smaller collections (that means less computational time for each processor) the corresponding time–saving due to pipelining on the query propagation phase (which is a part of the communication times) is more critical.

Another significant conclusion coming out from table 4 is that the combination of our data–balancing and pipelining techniques lead to the best possible experimental results for our system in all cases (under the restriction of R=1000).

The utilization measure raises up to 98% for 512 processors and the 8–WSJ collection, whereas the response time becomes 0.75 seconds for 512 processors and the 1–WSJ collection.

5.4 Multiple Queries Processing via Fat-trees

Beyond the extended experiments performed for the binary-tree algorithm of PFIRE we've also tried to obtain some realistic experimental results concerning our new fat-tree multiple-queries parallel algorithm ([20]). Briefly, the above algorithm, assuming an ideal fat-tree underlying interconnection network of P processors and a set of X concurrently arriving queries, it processes all the queries together and achieves excessively low total communication times (almost constant) by taking advantage of the high-capacity channels of the ideal fat–tree topology. Moreover (see [20]), the corresponding total processing times (both the amortized and the average waiting times) are proved, both analytically and experimentally over the TREC WSJ collection, to be substantially better than the ones offered by the binary–tree algorithm, provided that the two algorithms run on almost ideal fat-tree and binary-tree underlying networks. We achieve this in [20] by developing equivalently efficient embeedings (concerning the fat-tree we use a reduced version of the corresponding embeeding in order to have a fair comparison) of the two topologies over the 2-D mesh network of the GCel Parsytec machine (see [20]). The complete presentation of the algorithm as well as the detailed analysis of its' performance can be found in [20][2].

Via the following experiments (tables 6,7) we examine the case of using the full-version of our embeeding method ([20]) (which is quite unfair for the fat–tree algorithm due to the fact that its' high-capacity channels

[2]This reference can easily be found by contacting B. Mamalis to *mamalis@cti.gr*.

is impossible to be efficiently simulated by a 2-D mesh). In this case and for large values of P (e.g. greater than 128) the fat–tree algorithm can not offer better results than the binary–tree algorithm. However, for # of processors up to 64 all the corresponding experimental measures are clearly better. These experimental results are presented in table 6, whereas in table 7 the corresponding results for the binary–tree algorithm are given (with data balancing and pipelining) in order the relative comparisons to be easily performed[3]. Both the above tables are of the same type with tables 1,2 (and the value of R is equal to '1000') with the difference that here, we have used smaller in size collections (1/8, 1/4 and 1/2 WSJ) because of the smaller values of P.

Beyond the relative (not substantial, however clear) improvements offered by the fat–tree algorithm, the reader can easily notice that these improvements increase with the increase of P. This happens because as P increases the communication times of the fat–tree algorithm remain almost constant, whereas the communication times of the binary–tree algorithm increase logarithmically. A more clear view of the above observation is given in figure 4 where the utilization curves for both the algorithms over the 1/8–WSJ collection are presented (the distance between the two curves increase with the increase of P). However, if we had corresponding experimental results for values of P greater than 64 we would observe that the fat–tree utilization measures would decrease progressively and the utilization curve would start to go down sharply (due to the extra overheads that are caused by the full–version of our embeeding for large values of P).

6. References

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

[2] M. Lafazanis, B. Mamalis, P. Spirakis, B. Tampakas, and A. Tsakalidis, "FIRE: A Flexible Tool for Efficient Information Retrieval", RIAO '94 Conference, New York, October 11–13, 1994 (accepted for prototype demonstrations).

[3] K. Antonis, B. Mamalis, B. Papakostas, P. Spirakis, A. Stamoulis, B. Tampakas, "Effective Distributed Information Retrieval Techniques with the Vector Space Model", ISCA PDCS'96 Conference, September 25-27, Dijon, France, pp. 726-731, 1996.

[4] G. Salton, "Automatic Text Indexing Using Complex Identifiers", ACM SIGIR'88 Conference, pp.135–143, 1988.

[5] J.L. Fagan, "Automatic Phrase Indexing for Document Retrieval : An Examination of Syntactic and Non-Syntactic Methods", ACM SIGIR'87 Conference, pp. 91-99, 1987.

[6] G. Salton and McGill, "An Introduction to Information Retrieval", 2nd edition, McGraw–Hill, c1983.

[7] B. Frakes and R.B. Yates, "Information Retrieval: Data Structures and Algorithms", Addison–Wesley, c1992.

[8] C.J. Crouch, "A Cluster–Based Approach to Thesaurus Construction", ACM SIGIR'88, pp. 309-320, 1988.

[9] C.J. Crouch and B. Yang, "Experiments in Automatic Statistical Thesaurus Construction", ACM SIGIR'92 Conference, pp. 77–85, 1992.

[10] G. Salton, J. Allan, C. Buckley, "Approaches to Passage Retrieval in Full Text Information Systems", ACM SIGIR'93, pp. 49-58, 1993.

[11] M. Hearst, C. Plaunt, "Subtopic Structuring for Full Length Document Access", ACM SIGIR'93, pp. 59-68, 1993.

[12] D. Harman, "Relevance Feedback Revisited", ACM SIGIR'92 Conference, pp. 1-10, 1992.

[13] Frank Tiedt, "Parsytec GCel Supercomputer", Technical Report, Parsytec Computer GmbH, July 1992.

[14] G. Salton, et al, "A Vector Space Model for Automatic Indexing", Communications of the ACM, 18 : 613–620, 1975.

[15] Tong Li and V. Chiu, "X-Window Interface to SMART, an Advanced Text Retrieval System", 1992.

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

[17] J. Cringean, R. England, G. Manson, P. Willett, "Parallel text Searching in Serial Files Using a Processor Farm", ACM SIGIR'90, pp. 429-452, 1990.

[18] F. Grandi, P. Tiberio, P. Zezula, "Frame Sliced Partitioned Parallel Signature Files", ACM SIGIR'92, pp. 286-297, 1992.

[19] C. Stanfill, "Partitioned Posting Files: A Parallel Indexed File Structure for Information Retrieval", ACM SIGIR'90, pp. 413-428, June 1990.

[20] B. Mamalis, P. Spirakis, B. Tampakas, "Optimal High Performance Parallel Text Retrieval via Fat Trees", CTI Technical Report, Submitted to SIGIR'97 Conference.

[3] Relative comparisons to the experimental results of [20] can not be done due to the use of different settings (e.g the value of R, different versions of the embeeding etc.)