Parallel Computations in Information Retrieval

G. Salton*

D. Bergmark+

TR 80-439

*Department of Computer Science
Cornell University
Ithaca, New York 14853

+Office of Computer Services
Cornell University
Ithaca, New York 14853

September 1980

This study supported in part by the National Science Foundation under grant IST 79-09176.

Parallel Computations in Information Retrieval

G. Salton* and D. Bergmark+

Abstract

Conventional information retrieval processes are largely based on data movement, pointer manipulations and integer arithmetic; more refined retrieval algorithms may in addition benefit from substantial computational power.

In the present study a number of parallel processing methods are described that serve to enhance retrieval services. In conventional retrieval environments parallel list processing and parallel search facilities are of greatest interest. In more advanced systems, the use of array processors also proves beneficial. Various information retrieval processes are examined and evidence is given to demonstrate the usefulness of parallel processing and fast computational facilities in information retrieval.

---

* Department of Computer Science, Cornell University, Ithaca, New York 14853.
+ Office of Computer Services, Cornell University, Ithaca, New York 14853.
This study was supported in part by the National Science Foundation under grant IST 79-09176.

1. Conventional Information Retrieval

Information retrieval systems normally provide effective procedures for the storage, processing, and retrieval of information items in response to requests submitted by a population of users. Most conventional retrieval methods utilize the well-known inverted file technology in which an auxiliary index (the inverted index) is used in addition to the main file of information records. The index contains for each allowable search term a list of the record identifiers for all records identified by the corresponding term. The methods needed to identify the records responding to particular search requests thus are based principally on list processing operations involving the lists of record identifiers stored in the inverted index.

Consider as an example a typical Boolean query formulation such as ((TERM A AND TERM B) OR TERM C), implying that the user wishes to see items containing either the combination of terms A and B, or else term C. The following steps are needed to obtain the identifiers for all records to be retrieved from the main files:

a) the inverted index is used to obtain the list of record identifiers for all records containing term A;

b) the index is used to obtain the list of record identifiers for all records containing term B:

c) a list intersection operation furnishes the list of all items that are present on both the lists obtained under a) and b);

d) the inverted index is used to obtain the list of record identifiers for all records containing term C;

e) a list union operation generates the list of unique items contained either on the list obtained under c) or the list obtained under d).

A typical inverted index process is illustrated in the example of Fig. 1.

Following the list merge operations, the actual records corresponding to the identifiers contained on the final merged list must be retrieved from the main file. It should be noted that the full file of records is accessed only when records representing answers to the user queries must actually be retrieved. The operations needed to identify records that directly respond to a user query are all carried out by using the list processing methods in the inverted index as shown in Fig. 2.

The conventional retrieval operations described earlier are expensive to carry out for three main reasons:

a) the inverted index is typically very large because several tens of thousands of terms must be included in the index;

b) the lists of document identifiers corresponding to certain popular terms are long, and the list merge operations are therefore costly to carry out;

c) the number of records that may actually respond to a given user query may be large, implying that a substantial number of file accesses are required in the main document file to retrieve the records.

To respond to these potential inefficiencies, three types of improvements have been suggested that render the conventional retrieval operations more efficient: the use of a number of different search processors all operating simultaneously on different portions of the files; the use of comparison and merge networks to simplify the list processing operations; and finally, the use of so-called associative memories. All of these devices are based on the concept of parallelism: the operations are briefly explained in the next section.

2. Parallel Processing in Conventional Retrieval Systems

In many systems the files to be searched tend to become large. Thus in many operational retrieval systems, the number of stored records is of the order of several tens of thousands. In such cases a considerable effort is needed to find any particular item in the files. One obvious approach to reducing the search time (but not necessarily the cost) is to break up the files into a variety of subfiles--sometimes called cells--and to provide separate search facilities for each subfile. Typically, special purpose, back-end processors may be used, each charged with the management of one subfile. Furthermore, all the back-end processors are charged to operate in parallel on the data included in their particular subfile. A typical partitioned file organization of this type is shown in simplified form in Fig. 3. [1-5]

A partitioned file may be used to store either the inverted index or the main document file, or both. When the main file is partitioned, the search is speeded up by passing on to the individual back-end processors the record identifiers of all records satisfying the search logic. The several processors can then retrieve the corresponding records from their respective subfiles from where they are passed on to the host computer that actually transmits the results to the users. When the inverted index is also partitioned, the back-end processors are used to retrieve the lists of document references for the terms stored in each relevant subfile. The search logic consisting of the various list merge operations can then be carried out by connecting the back-end processors with each other; alternatively, the lists may be passed on to the general-purpose host computer for processing, or some other special-purpose device may be used.

When the number of search terms available in a retrieval system is large, the time required to find the addresses of the lists of document references corresponding to each search term may become substantial. In such a case it may be useful to maintain the list of search terms in a parallel-access memory, known as an associative memory, or associative store. In an associative memory all entries are searched in parallel and when a match is found between an input search term and a stored entry, the corresponding entry is immediately flagged. The basic system diagram for a computer using an associative store is shown in Fig. 4 and the corresponding search operations are illustrated in Fig. 5. [6-8]

In the example of Fig. 5, each line of the associative store contains a particular term used to identify the documents of a collection. The query term currently being processed is stored in the comparand register. A mask register may also be available that prevents certain character positions of the comparand register from taking part in the comparison operation. In general, all lines of the associative store are simultaneously compared with the entry in the comparand register, and proper matches are signalled in the response register. For the example of Fig. 5 a match exists with line 2 of the associative array. In inverted file processing, the response register might be used to indicate not only the location of a matching entry, but also the address of the corresponding list of document references.

The use of back-end processors and associative memories speeds up search operations involving large files. However, these devices do nothing directly for the list merging operation. Consider in particular the list processing operations previously illustrated in the example of Fig. 1. Assuming that ordered lists of document references are maintained in the inverted index for each search term, the following list merging operations are needed when two search terms are combined by the connectives AND, OR, and NOT, respectively:

a) A <u>OR</u> B:

the two ordered lists are merged into one ordered list, and one element of each duplicated entry is removed;

b) A <u>AND</u> B:

the two ordered lists are merged and all single entries are removed as well as one occurrence of each duplicated entry;

c) A <u>NOT</u> B:

the two ordered lists are merged and both occurrences of each duplicated entry are removed as well as all single entries originally on the B list.

The normal method for merging two ordered lists consists in comparing the first element from list A with the first element from B and transferring the smaller of the two elements--say the one from A--to the output. The next element from the A list is then compared with the first element from the B list, and the smaller one is chosen once again. The pairwise comparisons are then continued until the lists are exhausted. Assuming that each input list contains N/2 elements, N comparison operations are needed to merge the two lists.

The merging operation may be speeded up considerable by using several comparison units in parallel, each capable of comparing two input elements and identifying the smaller of the two. Thus N/2 separate comparison units may be used for lists of length N/2 to handle the first, second, third, and eventually the last entries from each of two ordered lists, respectively. The output from the first comparison stage can then be fed to additional comparison units constituting a second stage. Further stages of pairwise comparisons between certain list elements then follow until the final output represents a single ordered list. A sorting network of this type is outlined in Fig. 6. It is known that if the input consists of two ordered lists of N/2 entries, the number of stages needed in the ordering process when multiple comparison units are used is $(1+\lceil \log_2 N/2 \rceil)$ instead of N as before, and the number of comparison units needed is of order $N + N/2 \log_2 N/2$. [9-13]

The foregoing developments are all based on the same philosophy: search operations involving large files or lists of many elements may be speeded up by performing many operations in parallel. Such a solution, unfortunately, does not help in resolving a basic weakness of the conventional search and retrieval methodology: as the search queries become longer and involve more search terms, the amount of work needed to identify the matching records continually grows, because additional lists of document identifiers must then be processed. At the same time, the number of records to be retrieved from the main file becomes very hard to control: for Boolean queries involving ten or more search terms it is impossible to predict whether 0 records will eventually be retrieved, or 1,000. In either case, the user will presumably be dissatisfied. This suggests that a different retrieval philosophy may be needed not based on Boolean query formulations or on exact match strategies (where the stored documents are retrieved only when the terms assigned to the documents precisely match the query specification). This alternative retrieval environment is introduced in the next section.

3. The Vector Processing Model

Consider a collection of documents, each identified by a set of content identifiers, or terms. A given document, D_i, may then be represented by a term vector of the form

$$D_i = (d_{i1}, d_{i2}, \ldots, d_{it}) \tag{1}$$

where d_{ij} represents the importance factor, or weight, of the jth term assigned to D_i, and t is the total number of distinct terms assigned to the collection. A weight of 0 may be assumed for terms that are absent from a given vector, while a positive weight may be used for terms actually assigned to a vector.

Given two particular documents D_i and D_j, it becomes possible to compute a similarity coefficient between them based on the number of common terms in the vectors, and on the weight of the common terms. Typical similarity measures might be the inner product between the corresponding vectors or the cosine coefficient (expressions (2) and (3), respectively):

$$S(D_i,D_j) = \sum_{k=1}^{t} d_{ik}\, d_{jk} \tag{2}$$

$$S(D_i,D_j) = \frac{\sum_{k=1}^{t} d_{ik}\, d_{jk}}{\sqrt{\sum_{k=1}^{t} (d_{ik})^2 \cdot \sum_{k=1}^{t} (d_{jk})^2}} \; . \tag{3}$$

Both of these similarity measures produce a 0 zero value for vectors that have no common terms, and have a positive values when common terms exist. The maximum value of the cosine measure is equal to 1.

The computation of pairwise similarity measures between stored records suggests that documents whose vectors are sufficiently similar to each other be grouped to form classes of related documents. Consider the illustration of Fig. 7: if the distance on the plane is assumed to be inversely proportional to the vector similarity, then $S(D_i,D_j) >> (D_i,D_k)$ for the example of Fig. 7. In that case D_i and D_j might be grouped into a common class; D_k however would be excluded from the class. This leads to a clustered document collection of the type shown in Fig. 8 where certain items are grouped into common classes, or clusters. Each class may itself be identified by a class vector, also known as the centroid

$$C_p=(c_{p1},c_{p2},\ldots,c_{pt}) \tag{4}$$

where once again c_{pj} represents the weight of term j in the centroid for class p. The centroid could be defined as the average vector for all the documents in a given class. Assuming a class of m items

$$c_{pj} = \frac{1}{m} \sum_{D_i \in C_p} d_{ij} \,. \tag{5}$$

In the vector processing model, a given user query may also be represented as a vector of terms

$$Q_k = (q_{k1},q_{k2},\ldots,q_{kt}) \tag{6}$$

where q_{kj} represents the weight of the jth query term in Q_k. In these circumstances the complete retrieval operation may be reduced to a set of vector comparison operations as follows:

a) given a query Q_k, perform the similarity computation $S(Q_k,C_p)$ between Q_k and all cluster centroids C_p;

b) consider those clusters for which $S(Q_k,C_p)>T_1$ for some threshold value T_1; for all documents in the corresponding clusters, computer $S(Q_k,D_i)$;

c) arrange the documents in decreasing order of the similarity $S(Q_k,D_i)$ and present to the user all items such that

$S(Q_k, D_i) > T_2$ for some threshold value T_2.

By changing the values of the threshold T_1 and T_2, a variable number of items may be retrieved from the main file. Furthermore, the items may be presented to the user in decreasing order of presumed usefulness (that is, nearness to the query). The system again uses two principal files: the main document file, and an auxiliary file of cluster centroids that replaces the inverted index of the conventional system.

It is not possible in the present context to describe the vector processing model in greater detail. [14,15] What has been said so far should make it clear that the parallel search facilities described earlier in this study are applicable in a vector processing system as they are in the standard retrieval environment. Obviously, if several document vectors could be compared simultaneously to a given query vector, the retrieval operations would be speeded up. In addition, the need to manipulate the query, document, and centroid vectors-- for example, by computing similarity coefficients between many vector pairs-- suggests that additional improvements are obtainable by using efficient methods for performing the numerical computations. This possibility is considered in the remainder of this study.

4. Array Processors

Many areas of computer application are distinguished chiefly by the need for substantial computational power. For example, in signal processing, large quantities of data are received over external devices, such as radar or satellite equipment, that subsequently require processing and "cleaning up." In such circumstances, the need for fast internal computation becomes overwhelming. To respond to this demand special processors, known as array processors (AP) have been developed that provide very fast arithmetic facilities and work in conjunction with a general purpose computer (the host computer) to which they are attached. [16,17]

Array processors are often implemented as specialized, high-speed floating-point machines working in parallel with their host computer. No character manipulation or input output facilities are normally provided. The computational power of AP's is due to two main features:

a) parallel functional units: instead of including all arithmetic and logical functional of the processor in a simple "arithmetic and logical unit" as is done in standard computers, the various functions of the central processing unit are split up into separate functional units that can all function in parallel; thus in some array processors it is possible to perform an addition in the adder, and also a multiplication in the multiplier, and also a fetch operation to retrieve an item of data from memory, and also an instruction decoding operation; all of these operations can be carried out in parallel using separate functional units.

b) pipelined functional units: some units of the array processor are pipelined to speed up the processing of a single function, notably addition and multiplication; this means that a given operation is carried out in steps, or stages, in such a way that a given processing unit can effectively carry out several operations at the same time, provided each operation is in a separate stage. A pipelined processing unit, say a multiplier, consisting of three stages is shown schematically in Fig. 9. Three operations (multiplications) can in principle be carried out in the multiplier at the same time: the first multiplication that was started earliest is in stage 3 in the illustration of Fig. 9, the second multiplication started one stage later is in stage 2, and the third started most recently is in stage 1. After each time unit the pipelined processor advances by one stage; that is a new operation can be started in each unit of time if the pipeline is filled, the results of the operation being available three stages later.

Because of the limited set of functions provided, the cost of AP processing is inexpensive (typically $40 per hour) compared with the cost of a large standard computer (typically $1,000 per hour).

When an array processor is coupled to a general-purpose (host) computer as shown in Fig. 10, all input-output, program set-up, and data base operations are normally carried out by the host. Computational tasks can however be assigned to the AP after transfer by the host of relevant instructions and data into the array processor. The AP then executes its program while the host waits or performs other tasks unrelated to what is going on inside the AP. When the AP finishes its task, a "device interrupt" is sent to the host; the host then reads the results out of the AP, and processing continues.

Whether it pays to use an AP with a host computer depends on whether the savings obtained by executing a routine in the AP outweigh the costs of transferring programs and data between host and AP. The following factors appear important in this connection:

a) the data manipulations should be executable as floating-point arithmetic rather than as address, character, or integer manipulations;

b) the application should include long computations to justify the required host overhead and data transfer time;

c) the program to be executed should be small and the indexing requirements should be simple.

Information retrieval appears to furnish a poor application for AP's because of the large data base to be processed, and the many data transformation, as opposed to arithmetic, operations to be performed. On the other hand, it was seen earlier that the computational requirements are certainly not negligible in many information retrieval processes. Examples are the computation of similarity coefficients between vectors, and the generation of cluster centroids for clustered document collections. The parallel execution of one of these operations is covered in detail in the remainder of this study.

5. Vector Comparison Operations Using Array Processors

A typical information retrieval process consists of the following main operations: indexing, that is, assigning content identifiers and weights to the stored records; classification and file organization;

query formulation; searching and retrieving; query reformulation and search repetition, if necessary. All of these operations may be based on vector manipulations that could be carried out with array processors. [18] For present purposes, the single illustration involving information searching must suffice.

Consider a typical search operation. Assuming that a combination of processors is available consisting of a general-purpose host computer coupled to an array processor, the sequence of operations outlined in the flowchart of Fig. 11 might be used to search a clustered document collection. The assumption is that the files are stored in the host computer but that all vector comparisons (query-centroid and query-document matches) are carried out in the array processor.

It may be seen from Fig. 11 that the search operations consist of information transfers and of vector comparison operations of the form $S(Q_k, C_p)$ and $S(Q_k, D_i)$. Many different vector similarity measures are discussed in the literature. For present purposes, the cosine coefficient of expression (3) may be assumed as a standard. [14,15]

The computations of the cosine measure between two vectors--for example a query Q_k and a document D_i--may be broken down into two distinct parts:

a) the generation of the inner product $\sum_{\ell=1}^{t} q_{k\ell} d_{i\ell}$

b) the generation of the inverse norms of the vectors

$$(1/\sqrt{\sum_{\ell=1}^{t} (q_{k\ell})^2} \quad \text{and} \quad 1/\sqrt{\sum_{\ell=1} (d_{i\ell})^2}) \quad .$$

and the multiplication by the inverse norms.

Since the inverse norms represent constants for each vector, they can be computed in advance and stored with each corresponding term vector. This insures that the norms are available when needed.

The inner product computation consists of multiplications between vector elements, additions, and of course memory fetches to extract the required operands from storage. When an array processor is used to perform the operations it appears that several steps could be overlapped:

a) the multiplication of the jth vector elements from Q_k and D_i respectively, that is, $q_{kj} \cdot d_{ij}$

b) the addition of the (j-1)th product to the previous vector sum, that is

$$\sum_{\ell=1}^{j-2} q_{k\ell} \cdot d_{i\ell} + q_{k,j-1} \cdot d_{i,j-1}$$

c) the fetching from memory of the operands needed for the next product, that is, $q_{k,j+1}$ and $d_{i,j+1}$.

In order to describe the process in more detail it is necessary to introduce a specific format for storing the vectors in the array processor, and a particular array processor to carry out the operations. In principle, a <u>full vector</u> format may be used to store the document, centroid, and query vectors (expressions (1), (4) and (6)), where the kth vector element is used to store the weight of term k in the vector. In such a case, each vector is of length t, where t represents the total number of terms assignable in a particular indexing system. In a typical document retrieval environment t may be of the order of 10,000, but the number of terms actually present in a given vector may be of the order of 100. If a full vector format were used, up to 9,900 terms that are absent from a given vector would then be represented by weighting elements set equal to 0.

To avoid the storage of such long vectors most of whose elements are equal to 0, a <u>sparse vector</u> format may be used which includes only terms having a nonzero weight. Assuming that L nonzero term weights are present, a vector may then be represented by 2L vector elements as follows

$$(t_1,w_1;t_2,\ldots,t_L,w_L)$$

where t_i represents the index of the ith nonzero term and w_i represents the corresponding weight.

The array processor used for current purposes is the FPS 190-L consisting of a two-stage floating point adder and a three-stage floating point multiplier as shown in Fig. 12. At Cornell University, an IBM 370/168 acts as a general- purpose host computer in conjunction with the 190-L.

In the 190-L array processor, memory fetches from the main data memory can be started every other cycle, but an actual data item brought in from memory is available after three cycles only. Thus six cycles are needed to fetch two operands from the data memory, both data elements being available at the beginning of the seventh cycle. An addition and a multiplication can be started on every cycle, but the corresponding sum and product requires two and three cycles respectively for completion. The 190-L AP also includes a fast table memory in which memory fetches take only two cycles, instead of three for the data memory. A separate instruction memory is used to store the AP instructions.

On the 190-L array processor, one cycle of operations is performed every 167 nanoseconds, and as will be seen one complete loop for the inner product computation requires 4 cycles (667 nanoseconds). Various formats are usable to store the document and query vectors. For current purposes a sparse format is assumed for the document vectors stored in the data memory, and an expanded format for the queries stored in the fast table memory. In particular, a sparse document may appear as

$$D = (L+1, Dt_1, Dw_1, Dt_2, Dw_2, \ldots, Dt_L, Dw_L, 0, \frac{1}{\sqrt{\sum_{i=1}^{L} (Dw_i)^2}}),$$

where L+1 represents the number of nonzero term weights L plus 1, and the last term is the inverse norm required for the cosine computation. Each Dt_i designates the index, or column number of a term, and Dw_i is the corresponding nonzero term weight. An expanded query vector appears as

$$Q = (1/\sqrt{\sum_{i=1}^{t} (w_i)^2}, Qw_1, Qw_2, \cdots, Qw_t)$$

where Qw_i is a query term weight, and t is the total number of terms in the vocabulary. Most Qw_i's will appear as 0 in the expanded query format.

The only query weights of interest for the inner product are those corresponding to nonzero document weights. Hence a given document term index Dt_i can be used directly as an address to retrieve the corresponding query weight Qw (equal to $Q(Dt_i)$) from the fast table memory. The 0 stored in the document term vector following element Dt_L is used as the index to retrieve the zeroth element from the query vector, representing the inverse norm of the query $(1/\sqrt{\sum_{i=1}^{t} (Qw_i)^2})$. This can then be multiplied with the inverse norm for the document vector (the last element of D) as required for the cosine computation.

The basic four-cycle loop used to compute one step of the inner product is shown in Fig. 13. Three cycles are needed to fetch a document weight Dw_i from the data memory. The previous product $P_{i-1} = Q(Dt_{i-1}) \cdot Dw_{i-1}$ is started on cycle 4 to be ready two cycles later. The partial sum can then be initiated on the following cycle. (The notation SUM_{i-2} used in Fig. 13 stands for $\sum_{j=1}^{i-1} Qw_j \cdot Dw_j)$. A scratch pad memory also available on the AP is used as a loop counter: L+1 loops are needed to complete the computation.

The indexing operation used to retrieve a particular query term weight Qw_i, corresponding to a nonzero Dw_i, is superimposed on the inner product computation. The indexing operation is shown in detail in Fig. 14. Since the data memory of the AP stores floating-point numbers, the floating point representation of Dt_i must be transformed to fixed point notation before being used as an address to retrieve the corresponding Qw_i. This operation is outlined in Fig. 14.

It is not possible in the present context to completely evaluate the operations of the combined host-AP configuration used for information retrieval purposes. This requires detailed consideration of the complete retrieval process which must remain beyond the scope of the present study. It may suffice for present purposes to cite experimental timing and cost figures relating to the inner product computation only. [18,19]

It was seen earlier that about 2/3 of a microsecond (0.667 nanoseconds) are required per nonzero term for the inner product computation on the array processor. This compares with about 1.75 microseconds for the same action carried out on a 370/168 computer. The speed of the AP is offset by two kinds of overhead: first, the host processor overhead needed to decode the channel programs that transfer data and instructions between the host and the AP; and second the actual channel transfer times. For the combined 370/168-190L configuration, 7 milliseconds are needed to fill the AP data memory, 3 milliseconds to transfer results computed by the AP to the host machine, 6 milliseconds to transfer AP instructions to the AP instruction memory and to invoke the AP instructions, and finally 53 milliseconds to initialize the AP. The latter operation is required once for a given job when the AP is first turned on.

It is obvious from these figures that the extra cost of data and instruction transfer between host and AP must be offset by economies in the computations. For the retrieval application used as an illustration this appears relatively easy because the processing of a given query involves the generation of many vector correlations between query and centroid or document vectors. A sample chart appears in Fig. 15 reflecting the time needed to perform similarity computations between a 17-term query and 400 documents exhibiting an average of 155 nonzero terms. If the host time is assumed to be $1400/hour and the AP time is charged at $40/hour, the 400 correlations can be carried out at a total cost of 5.8 cents when the host operates alone; the host-AP combination costs less than one-tenth that amount (0.5167 cents) for that operation.

Complete timing studies are of course required to justify the use of array processing in a multi-user on-line retrieval environment.

REFERENCES

[1] S.Y.W. Su, Cellular Logic Devices: Concepts and Applications, Computer, Vol. 12, No. 3, March 1979, p. 11-25.

[2] G.P. Copeland, G.J. Lipovski, and S.Y.W. Su, The Architecture of CASSM: A Cellular System for Non-numeric Processing, Proceedings of the First Annual Symposium on Computer Architecture, Association for Computing Machinery, New York, December 1973, p. 121-125.

[3] P.J. Sadowski and S.A. Schuster, Exploiting Parallelism in a Relational Associative Processor, Proceedings of the Fourth Workshop on Computer Architecture for Non-numeric Processing, Association for Computing Machinery, New York, August 1978, p. 99-109.

[4] S.A. Schuster, H.B. Nguyen, E.A. Ozkarahan and K.C. Smith, RAP2-An Associative Processor for Data Bases and Its Applications, IEEE Transactions on Computers, Vol. C-28, No. 6, June 1979, p. 446-458.

[5] S.A. Schuster, H.B. Nguyen, E. A. Ozkarahan and K.C. Smith, RAP2-An Associative Processor for Data Bases, Proceedings of the Fifth Annual Symposium on Computer Architecture, Association for Computing Machinery, New York, April 1978, p. 52-59.

[6] C.R. DeFiore and P.B. Berra, A Quantitative Analysis of the Utilization of Associative Memories in Data Management, IEEE Transactions on Computers, Vol. C-23, No. 2, February 1979, p. 121-133.

[7] E.S. Davis, STARAN Parallel Processor Software, AFIPS Conference Proceedings, Vol. 43, AFIPS Press, Montvale, New Jersey, 1974, p. 16-22.

[8] J.A. Rudolph, A Production Implementation of an Associative Array Processor - STARAN, AFIPS Conference Proceedings, Vol. 41, Part 1, AFIPS Press, Montvale, New Jersey, 1972, p. 229-241.

[9] D.E. Knuth, The Art of Programming, Vol. 3, Searching and Sorting, Addison Wesley Publishing Company, Reading, Massachusetts, 1973, p. 224-230.

[10] L.A. Hollaar, A Design for a List Merging Network, IEEE Transactions on Computers, Vol. C-28, No. 6, June 1979, p. 406-413.

[11] W.H. Stellhorn, An Inverted File Processor for Information Retrieval, IEEE Transactions on Computers, Vol. C-26, No. 12, December 1977, p. 1258-1267.

[12] L.A. Hollaar and W.H. Stellhorn, A Specialized Architecture for Textual Retrieval, AFIPS Conference Proceedings, Vol. 46, AFIPS Press, Montvale, New Jersey, 1977, p. 697-702.

[13] L.A. Hollaar, Specialized Merge Processor Networks for Combining Sorted Lists, ACM Transactions on Data Base Systems, Vol. 3, No. 3, September 1978, p. 272-284.

[14] G. Salton, editor, The Smart Retrieval System--Experiments in Automatic Document Processing, Prentice Hall Inc., Englewood Cliffs, New Jersey, 1971.

[15] G. Salton, Dynamic Information and Library Processing, Prentice Hall Inc., Englewood Cliffs, New Jersey, 1975.

[16] A.L. Robinson, Array Processors: Maxi-Number Crunching for a Mini Price, Science, Vol. 203, January 12, 1979, p. 156-160.

[17] C.N. Winningstad, Scientific Computing on a Budget, Datamation, Vol. 24, No. 10, October 1978, p. 159-173.

[18] G. Salton, D. Bergmark, and A. Hanushevsky, Using Array processors in Information Retrieval, Technical Report, Computer Science Department, Cornell University, Ithaca, New York, 1980.

[19] D. Bergmark and A. Hanushevsky, Document Retrieval: A Novel Application for the AP, FPS User's Group Meeting, Los Angeles, California, 1980.

Term A : {12, 25, 36, 89, 125, 128, 215}

Term B : {11, 12, 17, 36, 78, 136, 215}

Term C : {11, 18, 36, 125, 132, 216}

a) Initial Lists of Document Identifiers from Inverted Index

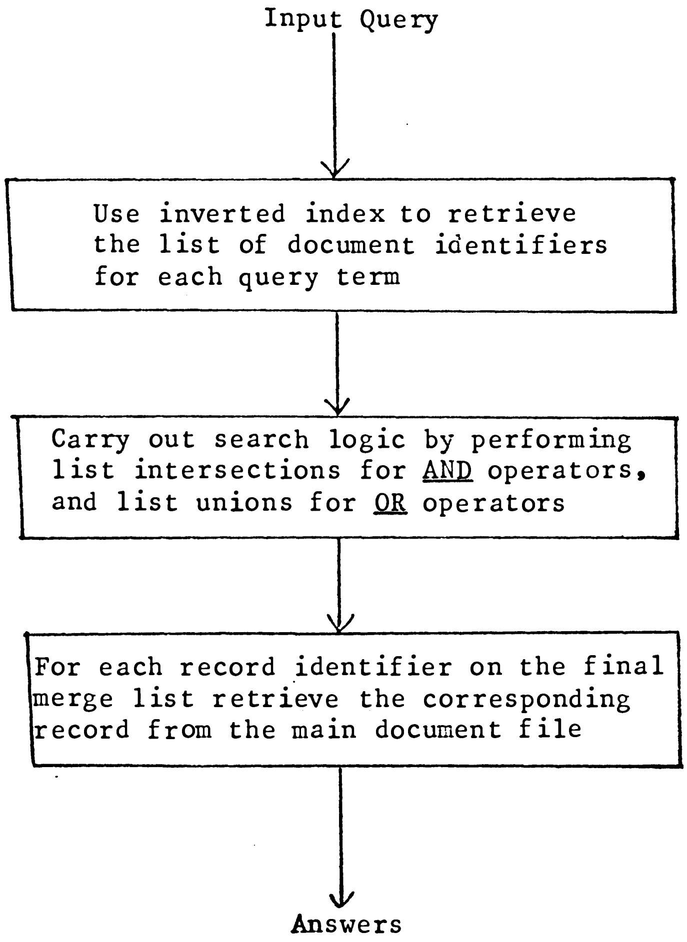
Term A AND Term B : {12, 36, 215}

((TERM A AND TERM B)
OR TERM C) : {11, 12, 18, 36, 125, 132, 215, 216}

b) List Intersection and List Union Operations

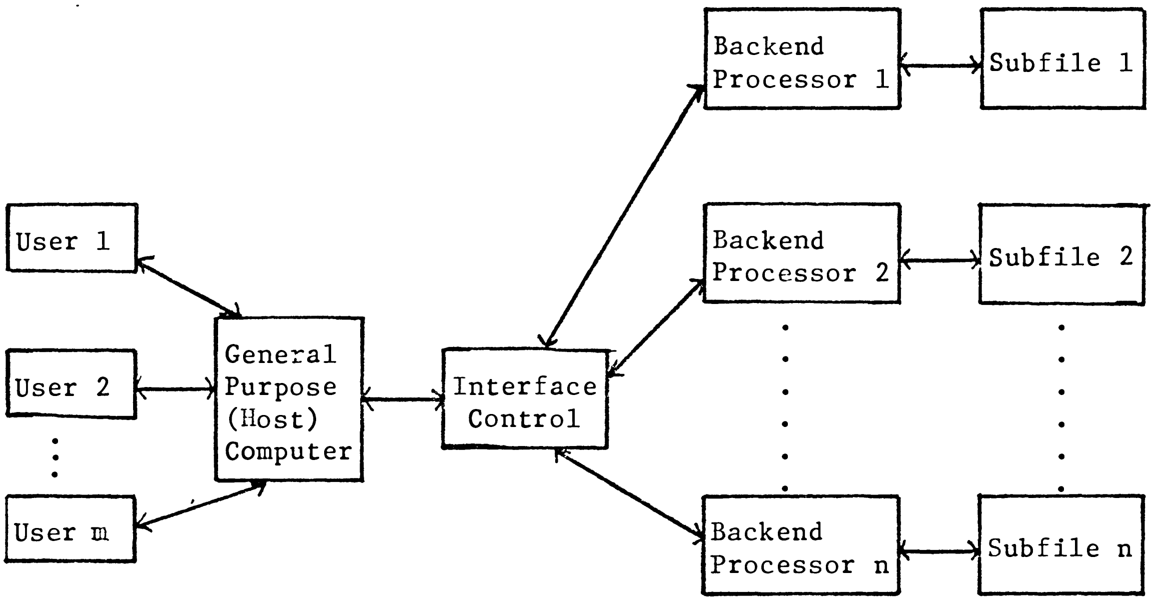
Example of Inverted File List Processing Operations

Fig. 1



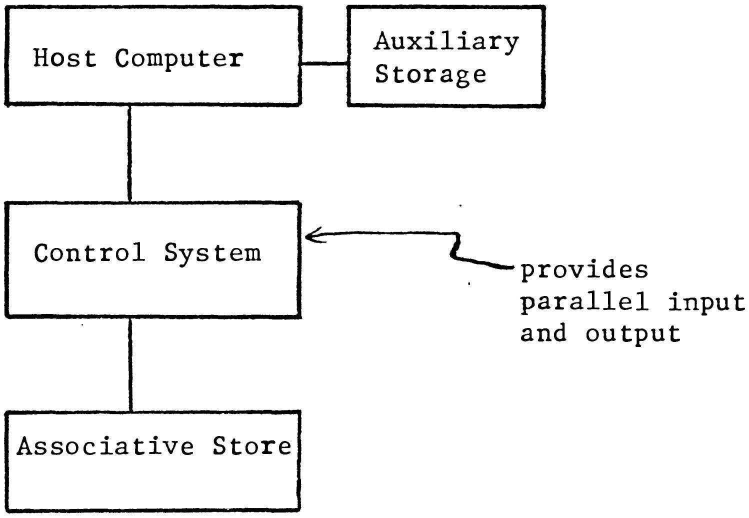
Conventional Inverted File Processing

Fig. 2



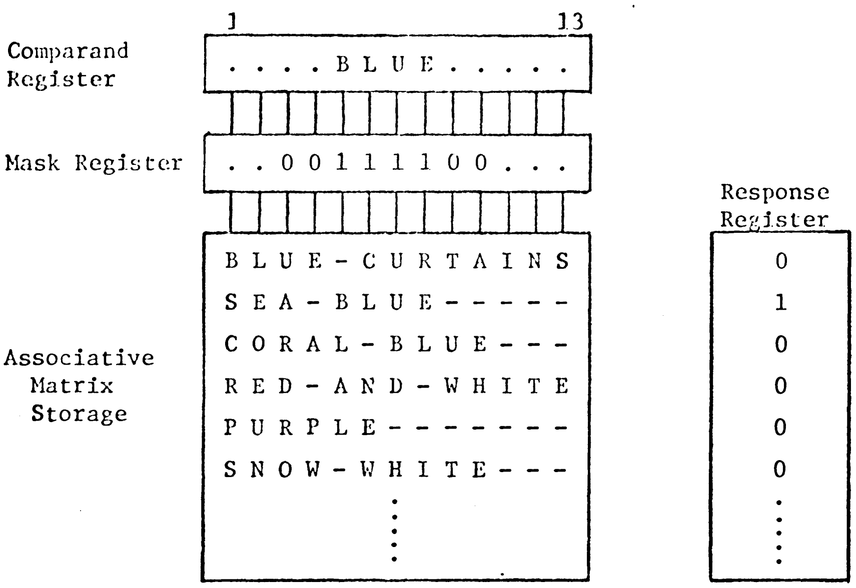
Use of Multiple Backend Search Processors

Fig. 3



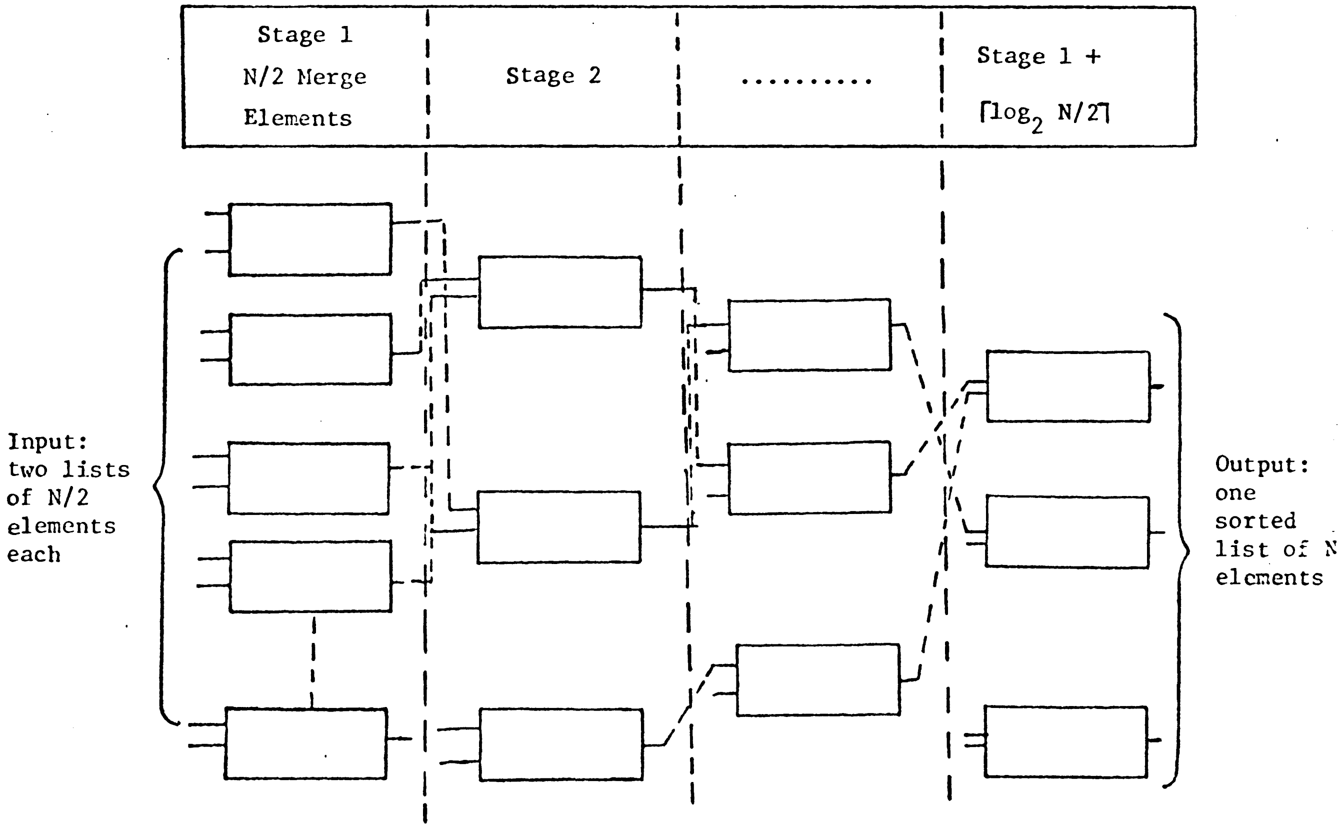
System Using Host Computer with Associative Store

Fig. 4



Parallel Associative Matching

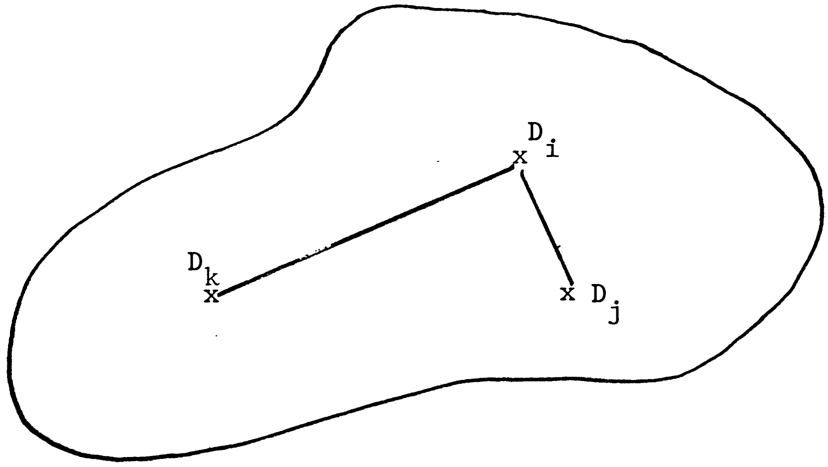
Fig. 5



Sample Merge Network

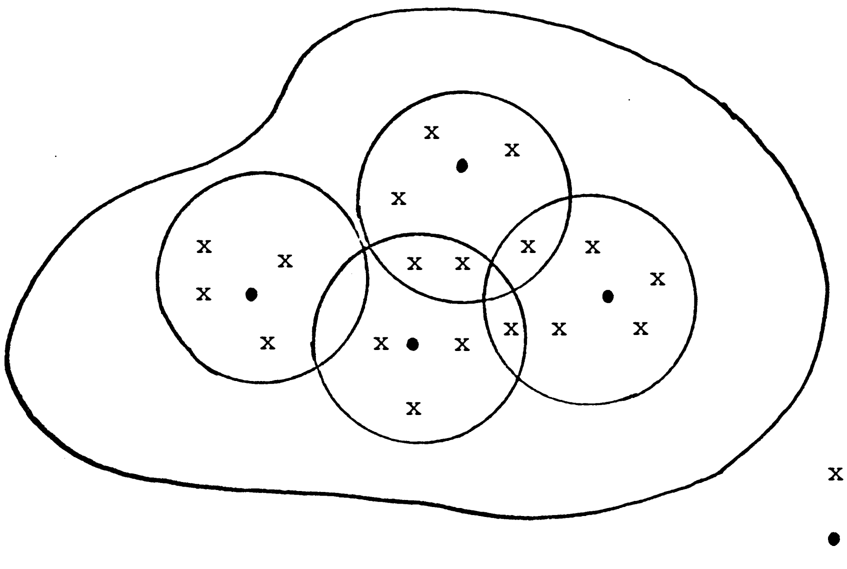
(N inputs, 1 + $\lceil \log_2 N/2 \rceil$ stages, order N + N/2 $\log_2$ N/2 comparison elements)

Fig. 6



Document Distances or Similarities

Fig. 7

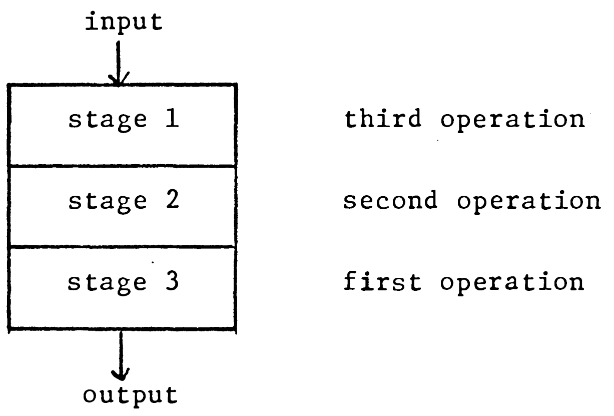


x individual document

• class centroid

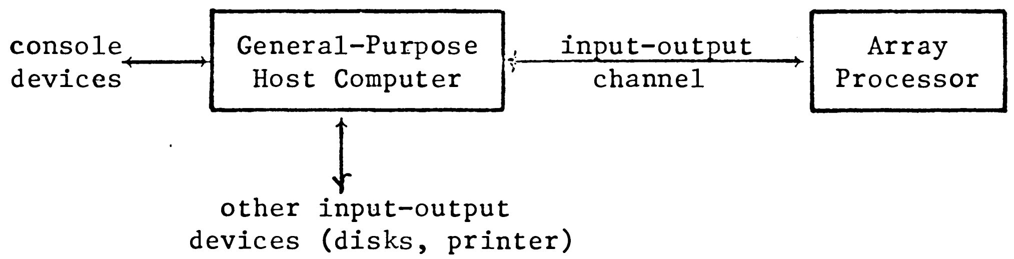
Clustered Document Collection

Fig. 8



Pipelined Processing Unit

Fig. 9



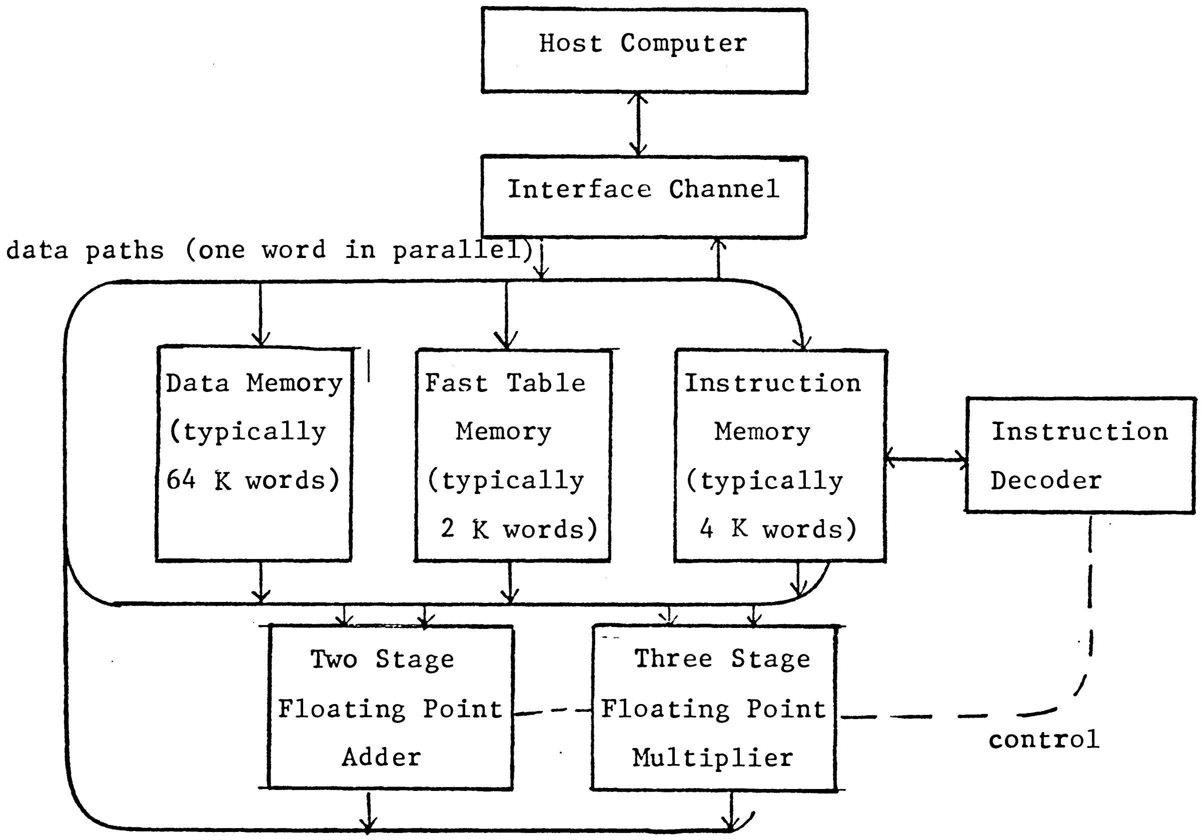
Simplified Diagram of Host-AP System

Fig. 10

Host Computer	Array Processor (AP)
1. User types in query which is transformed into a term vector and sent into the AP	idle
host idle (could perform other work)	2. The query vector is compared with the stored centroid vectors for the clustered documents and the best centroids are identified
3. The document vectors corresponding to the best centroids are sent to the AP	3. The AP starts comparison of query with some of the document vectors
4. Search results are obtained from the AP and corresponding documents retrieved from files	4. Query-document comparisons are carried out and identifiers for the most highly matching documents are sent to the host
5. Document citations are presented to the user and query may be reformulated	5. AP is initialized for a new search

Typical Search Process Using Host-AP Combination

Fig. 11



Typical Floating Point Array Processor

(constants, data and instruction; are kept in separate memories; multiplier and adder are pipelined; integer arithmetic, instruction decoder, adder and multiplier are separate functional units)

Fig. 12

One Loop

Functional Unit	Cycle 1	Cycle 2	Cycle 3	Cycle 4
Data Memory (documents)	Fetch Dw_i from next memory location			Dw_i now available; save it in register; Dw_{i-1} to multiplier
Adder			Start $SUM_{i-2} = SUM_{i-3} + P_{i-2}$	
Multiplier	Step 2 of product	Product $P_{i-2} = Q(Dt_{i-2}) \cdot Dw_{i-2}$ now ready		Start $P_{i-1} = Q_j \cdot Dw_{i-1}$
Table Memory (queries)		Fetch Q_j from table memory (Q_j is query weight for term i-1, that is $Q(Dt_{i-1})$)		Q_j now available for use in multiplier
Scratch Memory			Decrement loop counter	Repeat loop of index not negative

Basic 4 Cycle Loop to Carry Out Inner Product Computation

Fig. 13

One Loop

Functional Unit	Cycle 1	Cycle 2	Cycle 3	Cycle 4
Data Memory (documents)		Dt_i now available; send to adder to transform to integer format	Start fetch of Dt_{i+1} from next location in memory	
Adder		Fix Dt_i to integer format (previous SUM_{i-3} now ready)	Fixed Dt_i now ready (when new sum is started) store into scratch memory	
Table Memory (queries)		Start fetch of $Q(Dt_{i-1})$		
Scratch Memory	Load fixed Dt_{i-1} as address to initiate fetch of $Q(Dt_{i-1})$			

Indexing Operation Needed to Find Address
for Next Required Query Term

Fig. 14

Operation	Host alone	Host	+ AP
Send documents into AP memory	-	7.87	7.87
Perform similarity computation between one query and 400 document vectors	145	-	31.84
Return results from AP	-	5.40	5.40
Total time in milliseconds	145	13.27	45.11
Total cost in cents	5.80	0.5162	0.0005

Timing and Cost Figures for 400 Similarity Computations between Query and Document Vectors

(17 query terms, 155 nonzero terms per document)

Fig. 15