(Sync|Async)$^+$ MPI Search Engines

Mauricio Marin[1] and Veronica Gil Costa[2]

[1] Yahoo! Research, Santiago, University of Chile
[2] DCC, University of San Luis, Argentina

Abstract. We propose a parallel MPI search engine that is capable of automatically switching between asynchronous message passing and bulk-synchronous message passing modes of operation. When the observed query traffic is small or moderate the standard multiple managers/workers thread based model of message passing is applied for processing the queries. However, when the query traffic increases a round-robin based approach is applied in order to prevent from unstable behavior coming from queries demanding the use of a large amount of resources in computation, communication and disk accesses. This is achieved by (**i**) a suitable object-oriented multi-threaded MPI software design and (**ii**) an "atomic" organization of the query processing which allows the use of a novel control strategy that decides the proper mode of operation.

1 Introduction

The distributed inverted file is a well-known index data structure for supporting fast searches on Search Engines dealing with very large text collections [1–5, 7, 8]. An inverted file is composed of a vocabulary table and a set of posting lists. The vocabulary table contains the set of relevant terms found in the collection. Each of these terms is associated with a posting list which contains the document identifiers where the term appears in the collection along with additional data used for ranking purposes. To solve a query, it is necessary to get the set of documents associated with the query terms and then perform a ranking of these documents in order to select the top K documents as the query answer. In this paper we assume posting list items composed of pairs of document identifier and frequency in which the associated term appears in the given document.

The approach used by well-known Web Search Engines to the parallelization of inverted files is pragmatic, namely they use the document partitioned approach. Documents are evenly distributed on P processors and an independent inverted file is constructed for each of the P sets of documents. The disadvantage is that each user query has to be sent to the P processors which leads this strategy to a poor $O(P)$ scalability. Apart from the communication cost, sending a copy of every query to each processor increases overheads associated with large number of threads and disk operations that have to be scheduled. It can also present imbalance at posting lists level (this increases disk access and interprocessor communication costs). The advantage is that document partitioned indexes are easy to maintain since insertion of new documents can be

done locally and this locality is extremely convenient for the posting list intersection operations required to solve the queries (they come for free in terms of communication costs).

Another competing approach is the term partitioned index in which a single inverted file is constructed from the whole text collection to then distribute evenly the terms with their respective posting lists onto the processors. However, the term partitioned inverted file destroys the possibility of computing intersections for free in terms of communication cost and thereby one is compelled to use strategies such as smart distribution of terms onto processors to increase locality for most frequent terms (which can be detrimental for overall load balance) and caching. However, it is not necessary to broadcast queries to all processors. Nevertheless, the load balance is sensitive to queries referring to particular terms with high frequency and posting lists of differing sizes. In addition index construction and maintenance is much more costly in communication. However, this strategy is able to achieve $O(1)$ scalability.

Most implementations of distributed inverted files reported so far are based on the message passing approach to parallel computing in which we can find combinations of multithreaded and computation/communication overlapped systems. The typical case is to have in each of the P processing nodes a set of threads dedicated to receive queries and communicate with other nodes (threads) in order to produce an answer in the form of the top K documents that satisfy the query. However, it is known that threads can be potential sources of overheads and can produce unpredictable outcomes in terms of running time. Still another source of unpredictable behavior can be the accesses to disk used to retrieve the posting lists. Some queries can demand the retrieval of very large lists from secondary memory involving hundreds of disk blocks.

In that context the principle behind the findings reported in this paper can be explained by analogy with the classic round-robin strategy for dealing with a set of jobs competing to receive service from a processor. Under this strategy every job is given the same quantum of CPU so that jobs requiring large amounts of processing cannot monopolize the use of the CPU. This scheme can be seen as bulk-synchronous in the sense that jobs are allowed to perform a set of operations during their quantum. In our setting we define quantums in computation, disk accesses and communication given by respective "atoms" of size K where K is the number of documents to be presented to the user. We use a relaxed form of bulk-synchronous parallel computation [9] to process those atoms in parallel in a controlled (synchronous) manner with atoms large enough to properly amortize computation, disk and communication overheads.

For instance, for a moderate query traffic $q = 32$ and using a BSP library built on top of MPI (BSPonMPI http://bsponmpi.sourceforge.net/) we found this bulk-synchronous way of query processing quite efficient with respect to message passing realizations in MPI and PVM. See figure 1 that shows results for two text collections indexed using the document (D) and term (T) partitioned indexes. Notice that the technical details of the experiments reported in this paper are given in the Appendix. Also larger number of processors P implies larger

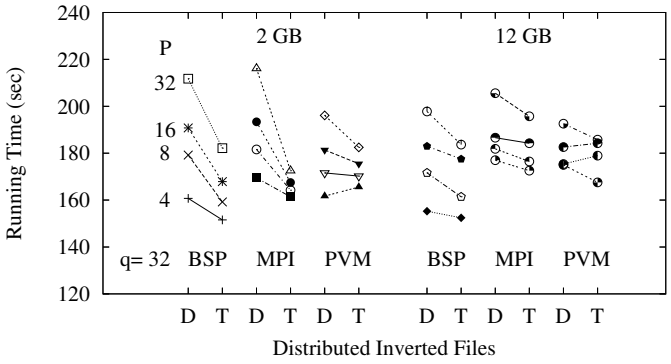

Fig. 1. Comparing BSP, MPI and PVM for inverted files under moderate query traffic.

running times because we inject in each processor a constant number of queries. This because the inter-processor communication cost is always an increasing function of P for any architecture. These results shows that our realizations of inverted files scale up efficiently because in each curve we duplicate the number of processors and running times increase modestly as $O(\log P)$.

However, we also observed that with a semi-synchronous MPI realization we were able to achieve similar performance to BSP. In this case we force every MPI processor to wait for P messages (one per processor) before delivering them to their target threads. The results are in figure 2 which shows other alternative implementations of inverted files where DB and TB represent bucketing strategies devised for improving load balance and T a bad (but in use) idea for implementing the term partitioned index. These results are evidence that in practice for this application of parallel computing it is not necessary to barrier synchronize all of the processors, it suffices to synchronize locally at processor level from messages arriving from the other processors.

Nevertheless the situation was quite different when we considered cases of low traffic of queries. The figure 3 shows results for two MPI realizations of the inverted files, namely an asynchronous message passing multi-threaded (Async) version and a non-threaded semi-bulk-synchronous MPI (Sync) realizations (details in the next section). This clearly makes a case for a hybrid implementation which is the discussion of this paper.

In the remainder of this paper we describe our proposal to make possible this hybrid form of parallel query processing using MPI. The rule to decide between one or another mode of operation is based on the simulation of a BSP computer. This simulation is performed on-the-fly as queries are received and send to processing.

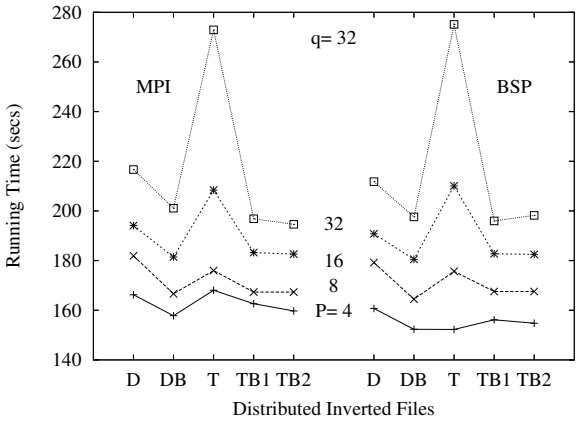

Fig. 2. Comparing BSP with a semi-synchronous MPI realization in which each processors waits to receive at least one message from all other processors before continuing query processing.

2 Round-Robin query processing

In this section we describe our algorithm for query processing and method for deciding between asynchronous and synchronous modes of operation. The parallel processing of queries is basically composed of a phase in which it is necessary to fetch parts of all of the posting lists associated with each term present in the query, and perform a ranking of documents in order to produce the results. After this, additional processing is required to produce the answer to the user. At the parallel server side, queries arrive from a receptionist machine that we call the *broker*. The broker machine is in charge of routing the queries to the cluster's processors and receiving the respective answers. It decides to which processor to route a given query by using a load balancing heuristic. The particular heuristic depends on the approach used to partition the inverted file. Overall the broker tends to evenly distribute the queries on all processors.

The processor in which a given query arrives is called the *ranker* for that query since it is in this processor where the associated document ranking is performed. Every query is processed using two major steps: the first one consists on fetching a K-sized piece of every posting list involved in the query and sending them to the ranker processor. In the second step, the ranker performs the actual ranking of documents and, if necessary, it asks for additional K-sized pieces of the posting lists in order to produce the K best ranked documents that are passed to the broker as the query results. We call this *iterations*. Thus the ranking process can take one or more iterations to finish. In every iteration a new piece of K pairs (doc_id, frequency) from posting lists are sent to the ranker for every term involved in the query. In this scheme, the ranking of two or more queries can take place in parallel at different processors together with the fetching of K-sized pieces of posting lists associated with other queries.

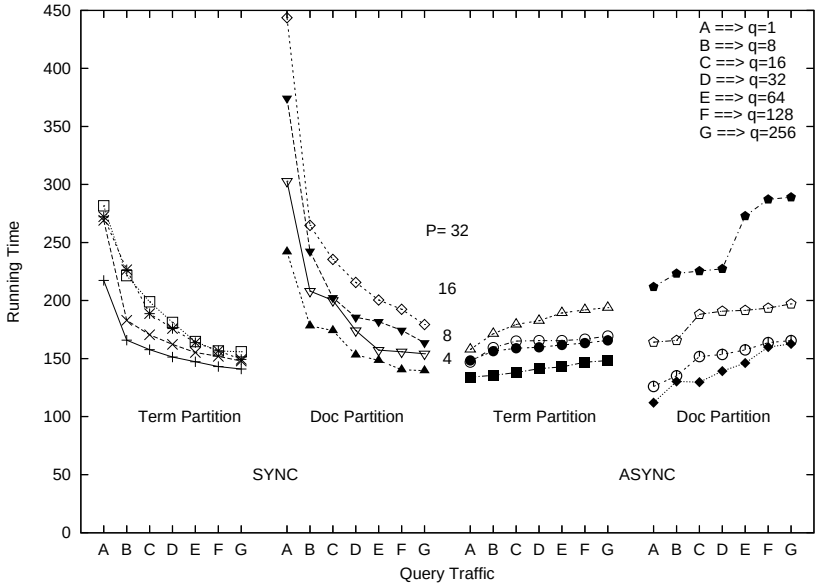

Fig. 3. Comparing MPI under bulk-synchronous (SYNC) and asynchronous (ASYNC) modes of operation for the term and document partitioned inverted files under different query traffics of q queries per processor per unit time for 32, 16, 8 and 4 processors.

We use the vectorial method for performing the ranking of documents along with the filtering technique proposed in [6]. Consequently, the posting lists are kept sorted by frequency in descending order. Once the ranker for a query receives all the required pieces of posting lists, they are merged into a single list and passed throughout the filters. If it happens that the document with the least frequency in one of the arrived pieces of posting lists passes the filter, then it is necessary to perform a new iteration for this term and all others in the same situation. We also provide support for performing the intersection of posting lists for boolean AND queries. In this case the ranking is performed over the documents that contain all the terms present in the query.

The synchronous search engine is implemented on top of the BSP model of parallel computing [9] as follows. In BSP the computation is organized as a sequence of *supersteps*. During a superstep, the processors may perform computations on local data and/or send messages to other processors. The messages are available for processing at their destinations by the next superstep, and each superstep is ended with the barrier synchronization of the processors. The underlying communication library ensures that all messages are available at their destinations before starting the next superstep.

Thus at the beginning of each superstep the processors get into their input message queues both new queries placed there by the broker and messages with pieces of posting lists related to the processing of queries which arrived at previous supersteps. The processing of a given query can take two or more supersteps

to be completed. All messages are sent at the end of every superstep and thereby they are sent to their destinations packed into one message per destination to reduce communication overheads.

Query processing is divided in "atoms" of size K, where K is the number of documents presented to the user as part of the query answer. These atoms are scheduled in a round-robin manner across supersteps and processors. The asynchronous tasks are given K sized quantums of processor time, communication network and disk accesses. These quantums are granted during supersteps, namely they are processed in a bulk-synchronous manner. As all atoms are equally sized then the net effect is that no particular task can restrain others from using the resources. During query processing, under an observed query traffic of $Q = q\,P$ queries per unit time with q per processor per superstep, the round-robin principle is applied as follows. Once Q new queries are evenly injected onto the P processors, their processing is started in iterations as described above. At the end of the next superstep some queries, say n queries, all requiring a single iteration, will finish their processing and thereby at the following superstep n new queries can start their processing. Queries requiring more iterations will continue consuming resources during a few more supersteps.

In each processor we maintain several threads which are in charge of processing the K-sized atoms. We use LAM-MPI so we put one thread to perform the inter-processors message passing. This thread acts as a scheduler and message router for the main threads in charge of solving the queries. We use the non-blocking message passing communication primitives. We organized our C++ code into a set of objects, among them we have the object called "processor" which is the entry point to all the index methods and ranking. The crucial point here is that all threads have access to this object and concurrency conflicts are avoided by keeping in thread's local memory the context of each queries they are in charge of. When the search engine switches to bulk-synchronous operation all threads are put to sleep on condition variables and the main thread takes control of processing sequentially the different stages of queries during supersteps.

When the search engine is operating in the asynchronous mode it simulates the operation of a BSP machine. This is effected every N_q completed queries per processor as follows. During the interval of N_q queries each processor of the asynchronous machine registers the total number of iterations required by each query being solved. The simulation of a BSP computer for the same period can be made by assuming that q new queries are received in each superstep and processor. For this period of Δ units of time, the observed value of q can be estimated in a very precise manner by using the G/G/∞ queuing model. Let S be the sum of the differences [DepartureTime - ArrivalTime] of queries, that is the sum of the intervals of time elapsed between the arrival of the queries and the end of their complete processing. Then the average q for the period is given by S/Δ. This because the number of active servers in a G/G/∞ system is defined as the ratio of the arrival rate of events to the service rate of events (λ/μ). If n queries are received by the processor during the interval Δ, then the arrival rate is $\lambda = n/\Delta$ and the service rate is $\mu = n/S$.

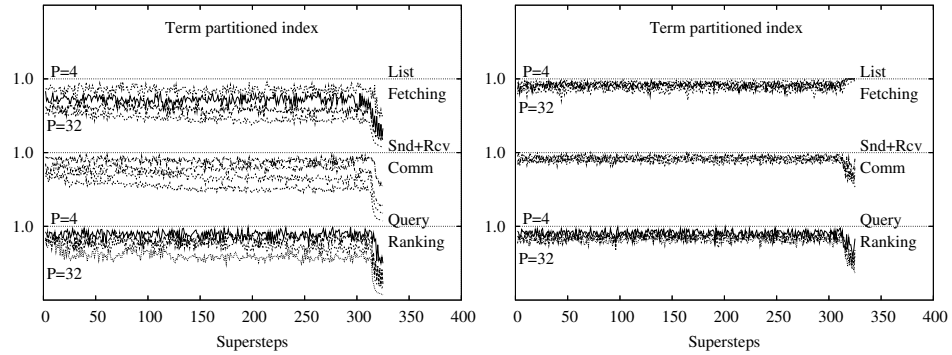

Fig. 4. Predicted SYNC efficiencies in disk accesses, communication and query ranking. Figure [left] is a case in which the query traffic is very low ($q = 1$) and figure [right] is a case of high traffic ($q = 128$). These extreme cases explains the performance of the SYNC term partitioned index in figure 3.

In addition the processors maintain the number of "atoms" of each type processed during the interval of running time. The efficiency metric is used to determine when to switch from one mode of operation to the another. For a metric x this is defined as the ration average(x)/maximum(x) both values taken over all processors and averaging across supersteps. The search engine switches to bulk-synchronous mode when efficiencies in ranking, communication and list-fetching are over 80%. Below that the asynchronous message passing mode is used. We have found that this simulation is accurate as a predictor of performance. For instance, this simulation predicts the efficiencies shown in figure 4 which are consequent with the bad and good performances observed in figure 3 for the term partitioned index for the same experiments in both cases.

3 Conclusions

We have presented a method and a MPI-based implementation to allow a search engine to dynamically switch its mode of parallel processing between asynchronous and bulk-synchronous message passing. This is achieved by dividing the tasks involved in the processing of queries into K-sized single-units and interleaving their execution across processors, network communication and disk-accesses. The glue between the two modes of operation is the simulation of a bulk-synchronous parallel computer. Our experiments show that this simulation is quite accurate and independent of the actual mode of operation of the search engine, be it under low or high traffic of queries.

Appendix

We use two text databases, a 2GB and 12GB samples of the Chilean Web taken from the `www.todocl.cl` search engine. The text is in Spanish. Using this collection we generated a 1.5GB index structure with 1,408,447 terms. Queries were selected at random from a set of 127,000 queries taken from the todocl log. The experiments were performed on a cluster with dual processors (2.8 GHz) that use NFS mounted directories. In every run we process 10,000 queries in *each* processor. That is the total number of queries processed in each experiment reported in this paper is 10,000 P. For our collection the values of the filters C_{ins} and C_{add} of the filtering method described in [6] were both set to 0.1 and we set K to 1020. On average, the processing of every query finished with $0.6K$ results after 1.5 iterations. Before measuring running times and to avoid any interference with the file system, we load into main memory all the files associated with queries and the inverted file.

References

1. C. Badue, R. Baeza-Yates, B. Ribeiro, and N. Ziviani. Distributed query processing using partitioned inverted files. *Eighth Symposium on String Processing and Information Retrieval (SPIRE'01)*, pages 10–20, Nov. 2001.
2. F. Cacheda, V. Plachouras, and I Ounis. Performance analysis of distributed architectures to index one terabyte of text. In *In S. McDonald and J. Tait, editors, Proc. ECIR European Conf. on IR Research*, pages 395–408, Sunderland, UK, April 2004.
3. B. S. Jeong and E. Omiecinski. Inverted file partitioning schemes in multiple disk systems. *IEEE Transactions on Parallel and Distributed Systems*, 16(2):142–153, 1995.
4. A.A. MacFarlane, J.A. McCann, and S.E. Robertson. Parallel search using partitioned inverted files. In *7th International Symposium on String Processing and Information Retrieval*, pages 209–220. (IEEE CS Press), 2000.
5. W. Moffat, J. Webber, Zobel, and R. Baeza-Yates. A pipelined architecture for distributed text query evaluation. *Information Retrieval*, October 5 2006.
6. M. Persin, J. Zobel, and R. Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *Journal of the American Society for Information Science*, 47(10):749–764, 1996.
7. B.A. Ribeiro-Neto and R.A. Barbosa. Query performance for tightly coupled distributed digital libraries.(acm press). *Third ACM Conference on Digital Libraries*, pages 182–190, 1998.
8. C. Stanfill. Partitioned posting files: a parallel inverted file structure for information retrieval. In *13th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 413–428, Brussels, Belgium, 1990.
9. L.G. Valiant. A bridging model for parallel computation. *Comm. ACM*, 33:103–111, Aug. 1990.