Improving Search Engines Performance on Multithreading Processors

Carolina Bonacic[1], Carlos Garcia[1], Mauricio Marin[2], Manuel Prieto[1], Francisco Tirado[1], and Cesar Vicente[1] *

[1] Depto. Arquitectura de Computadores y Automática
Universidad Complutense de Madrid
Contact-email: cbonacic@fis.ucm.es, {garsanca, mpmatias, ptirado}@dacya.ucm.es

[2] Yahoo! Research Santiago
University of Chile
mmarin@yahoo-inc.com

Abstract. In this paper we present strategies and experiments that show how to take advantage of the multi-threading parallelism available in Chip Multithreading (CMP) processors in the context of efficient query processing for search engines. We show that scalable performance can be achieved by letting the search engine go synchronous so that batches of queries can be processed concurrently in a simple but very efficient manner. Furthermore, our results indicate that the multithreading capabilities of modern CMP systems are not fully exploited when the search engine operates on a conventional asynchronous mode due to the moderate thread level parallelism that can be extracted from a single query.

1 Introduction

The algorithmic design and implementation of current Web Search Engines is based on the asynchronous message passing approach to parallel computing in which each newly arriving query is serviced by an independent thread in a classical multiple masters/slaves scheme. Typical facilities for parallel query processing at data centers are composed of a few thousand Linux boxes forming clusters of computers.

On the other hand, the amount of work demanded by the solution of queries follows the so-called Zipf's law which in practice means that some queries, in particular the ones composed of most popular terms, can demand large amounts of processing times whereas others containing less frequent terms can require a comparatively much smaller processing time.

* This work has been partially supported by the research contracts CICYT-TIN 2005/5619, CYTED-506PI0293, FONDECYT 1060776 and Ingenio 2010 Consolider CSD2007-20811. We also thank Sun Microsystems and the AulaSun of the UCM for their support.

Thus under this asynchronous approach and hardware latencies a given query can easily restrain smaller queries by consuming comparatively larger amounts of resources in processor cycles, disk and inter-processors network bandwidths.

However, we have found that a careful design of the major steps involved in the processing of queries can allow its decomposition in such a way that we can let every query share the cluster resources evenly in a round-robin manner [8, 9]. We have observed that this scheme can be particularly useful in preventing unstable behavior under unpredictable variations in the query traffic arriving to the search engine.

In particular, we have observed for the standard asynchronous method of query processing that sudden peaks in the query traffic can be very detrimental to overall performance due to the Zipf's law distribution of the workload per query. We have also observed that the round-robin method of query processing solves this problem efficiently. We have validated this claim through extensive experimentation by running actual query logs upon actual 1TB samples of the Web. Nevertheless, our experiments have been performed on standard machines with coarse-grain threads implemented by Posix software running on clusters supporting the distributed memory model and the MPI message passing communication library.

Having said that, it is clear that this discussion is only valid at a macroscopic level in terms of "heavy" threads and operations for query processing in a sharing nothing model for data distribution. However, state of the art computer architectures integrate facilities for light threads and shared memory, which are available to the programmer in the form of efficient realizations of the *OpenMP* model of parallel computing. In fact, future improvements in processor performance will predominantly come from Thread Level Parallelism, rather than from increasing clock frequency or processor complexity [2]. In this regard, we think that it is an interesting research problem to validate the above claims in the context of these new architectures and if not, explore new optimizations to achieve efficient performance under this new setting.

In this paper, we provide a first step in this direction by studying different realizations of standard and round-robin search engines implemented upon a state-of-the art Chip Multithreading system [11] and its respective *OpenMP* realization. As experimental platform we have chosen a Sun Microsystems' UltraSPARC T1 processor – code-named as Niagara [6] and marketed by Sun as *CoolThreads* technology – since it symbolizes the recent shift to CMP in the server market and presents a radical new approach to enable throughput computing and scalability with low power consumption.

For programming purposes the T1 can be seen as a set of logical processors that share some resources. Consequently, one may think that parallelization schemes targeted for other shared-memory multiprocessors, such as SMP systems, are also good candidates for this processor. However, the sharing of resources introduced on the T1 for increasing utilization may cause serious bottlenecks and hence, strategies that are appropriate for these machines may be

inappropriate or less effective for the T1. One of the goals motivating this study is to revise the implementation of parallel search engines in this light.

The rest of this paper is organized as follows: Section 2 and 3 describe our search engine and the experimental framework respectively. Section 4 presents our parallel proposals and Section 5 shows some performance results. Finally the paper ends with some conclusions and hints for future research.

2 Search engine overall description

2.1 Distributed inverted file

Web Search Engines use the inverted file data structure to index the text collection and speed up query processing. A number of papers have been published reporting experiments and proposals for efficient parallel query processing upon inverted files which are distributed on a set of P processor-memory pairs [1, 3, 4, 7–10, 14]. It is clear that efficiency on clusters of computers is only achieved by using strategies devised to reduce communication among processors and maintain a reasonable balance of the amount of computation and communication performed by the processors to solve the search queries.

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 *ids* associated with the query terms and then perform a ranking of these documents so as to select the top K documents as the query answer.

Current search engines use the document partitioned approach to distributing the inverted file on a set of P processors. In this case, the document collection is evenly distributed at random on the processors and an inverted file is constructed in each processor considering only the documents stored in the processor. Solving a query involves to (a) place a copy of it in each processor, (b) let each processor calculate their local top K results and (c) make a merge of all results to select the global top K results.

Query operations over parallel search engines are usually read-only requests upon the distributed inverted file. This means that one is not concerned with multiple users attempting to write information on the same text collection. All of them are serviced with no regards for consistency problems since no concurrent updates are performed over the data structure. Insertion of new documents is effected off-line.

2.2 Organizing query processing

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 (where for the scope of this paper each processor is a chip-multiprocessor node of the cluster) and receiving the respective answers. It decides to which processor routing 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.

More in detail, 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. This paper is concerned with the fetching+ranking part. We are interested in situations where it is relevant to optimize the query throughput.

A relevant issue for this paper is the way we organize query processing upon the piece of inverted file stored in each processor. We basically apply the combination of two strategies we have devised to efficiently cope with hardware resource contention among queries and dynamic variations in the query traffic:

- **Round robin query processing**. We let queries to use a fixed quantum of computation, communication and disk access before granting the resources to another query in a round-robin fashion.
- **Operation mode**. We dynamically switch the mode of operation of the search engine between the asynchronous and synchronous message passing modes of parallel computation in accordance with the observed query traffic.

In the following subsection we describe both strategies in detail.

2.3 Iterative ranking and round-robin query processing.

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 iteratively using two major steps:

- **Fetching**. 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 essence, the *ranker* sends a copy of every query to all other P nodes. Next, all nodes send K/P pairs (*doc_id, frequency*) of their posting lists to the *ranker* which performs the first iteration of the documents ranking process.
- **Ranking**. 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 use the vectorial method for performing the ranking of documents along with the filtering technique proposed in [12]. 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 less 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.

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 each term involved in the query. This concept of iteration is essential to distribute and allocate system resources to the queries in a round-robin fashion: the quantum comes from the fact that we let queries work on chunks of posting lists of size K and organize document ranking in iterations.

2.4 Operation Mode

As mentioned above, we dynamically switch the mode of operation in accordance with the query traffic observed.

- **Asynchronous mode**. Low query traffic triggers an asynchronous mode in which each query is serviced by a unique *master* thread in charge of processing the query. This *master* thread can communicate with P other *slave* threads, each located in one of the P cluster nodes.
- **Synchronous mode**. High query traffic triggers a mode in which all active threads are blocked and a single thread takes the control of query processing by grouping queries in batches and processing them sequentially. In this case messages are buffered in all cluster nodes and sent out at the end of the current batch being processed, point at which all processors are barrier synchronized. Better utilization of system resources of this mode comes from the fact that overheads such as thread scheduling and synchronization cost are reduced significantly and communication is performed in bulk.

3 Experimental framework. Computing platform and data set

As experimental platform, we have chosen a Sun Microsystems' UltraSPARC T1 processor, whose main features are summarized in Table 1. Initially codenamed as *Niagara*, the T1 is a special-purpose CMP designed by Sun for the server market. It is available with four, six or eight CPU cores, and each core allows for the execution of four threads concurrently. Essentially, T1 cores are fine-grain multithreading (FGM) processors [5] that switch between threads of execution on every cycle for hiding the inefficiencies caused by long operational latencies such as memory accesses [13]. Single thread applications will perform better on traditional processors, but multithreaded workloads may benefit from this architecture: each thread is slower but this architecture yields better use of the processor's resources and potentially a higher overall throughput.

In our implementations, thread level parallelism has been exploited by means of the *OpenMP* standard, which is supported by Sun's native compilers.

3.1 Fixed-Point ranking

The UltraSPARC-T1 processor has a limited floating-point capability since it only provides one floating-point unit to support all 8 cores on the chip, i.e.

Table 1. Main features of the target computing platform.

Processor	SUN UltraSPARC-T1 8 core processor (1.2GHz) (4-way fine-grain multithreading core)	
	L1 Cache (per core)	16+8 KB (instruction+data) 4-way associative, LRU
	L2 Unified Cache	3MB (4Banksx768KB) 12-way associative, pseudo-LRU
Memory	16 GBytes (4x4GBytes) DIMMS 533 MHz DDR2 SDRAM	
Operating System	SunOS 5.10 (Solaris 10) for UltraSparcT1	
Sun C/C++ Compiler v5.8 Switches	-fast -xarch=v9 -xipo=2 Parallelization with OpenMP: -xopenmp=parallel	

only one thread can use it at a time. Furthermore, even if just one thread uses the floating-point unit, there is a 40 cycle penalty to access the unit. Most commercial applications have little or no floating-point content so it is not a major handicap. However, in our target application, one of the most costly phases is the *ranker* process, which uses floating-point arithmetic to classify the most relevant documents for a query.

To overcome this potential bottleneck, we have modified the *ranker* to avoid floating-point arithmetic. Our implementation uses a 32-bit fixed-point data representation for holding the *appearance frequency* in the posting lists, instead of the conventional floating-point representation, and performs computations using a customized fixed-point library that takes advantage of the T1 integer ALUs – there is an integer ALU per core –. The overhead introduced in the fixed-point version by overflow checking is around 20 to 40%, but the large penalties introduced by floating-point operations (the floating-point *sqrt* takes thousand of cycles) compensate this cost.

Figure 1 illustrates the potential benefits of this optimization. It shows the scalability of a synthetic benchmark that tries to mimic the kind of computations performed by the ranking process – it mixes different arithmetic operations such as divisions, multiplications, logarithms and square roots –. As expected, the performance of the floating-point version is really poor but the fixed-point counterpart scales reasonable well.

3.2 DataSet Inputs

All the results of this work have been obtained using a Chilean Web database sample taken from *www.todocl.cl*. The index structure contains around 1 million Spanish terms – 1.5 GBytes in size –. Queries have been selected randomly from a set of 127.000 queries extracting from *todocl* logs.

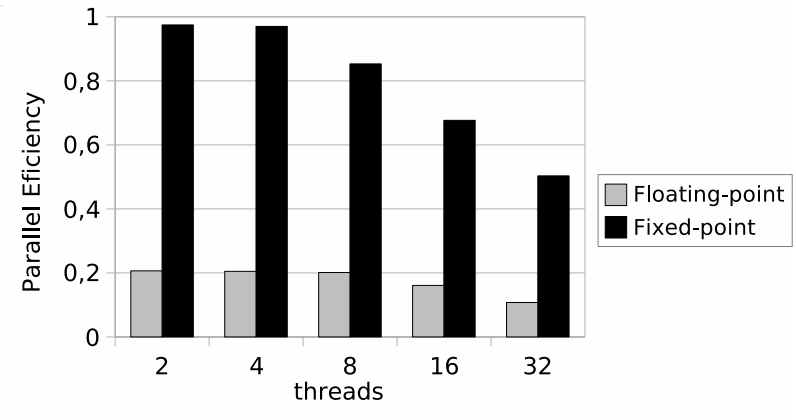

Fig. 1. Parallel efficiency of a synthetic benchmark that tries to mimic the kind of computations performed by the ranking process using either floating-point or fixed point arithmetic.

4 Parallel Scheme

As mentioned above, the availability of chip multithreading architectures introduces a new scenario in which thread-level parallelism becomes the key for achieving performance. In this regard, a critical issue here is the operation mode of the search engine since it strongly influences the way in which thread level parallelism can be extracted:

- **Synchronous mode**. Under high query traffic, batching the queries and letting the search engine go synchronous introduces regular coarse-grain parallelism. This parallelism can be easily translated into thread level parallelism in a simple manner by running queries into separate threads. Thread management overheads are relatively small at the expense of synchronization cost. The question here is whether this coarse-grain parallelization fits well with CMP characteristics.
- **Asynchronous mode**. Under low query traffic, it is necessary to resort to other sources of parallelism. The question here is whether intra-query parallelism is high enough to be exploited efficiently on CMP architectures.

In the following subsection we describe both approaches more precisely.

4.1 Synchronous mode

The coarse-grain parallelism introduced by the synchronous mode can be easily expressed by means of *OpenMP* directives using conventional query distribution schemes. However, the similarities amongst the different threads – they execute

the same code with just a different query – may cause contention for the shared resources of the T1, especially cache space and memory bandwidth. We have tried to minimize these penalties with a data distribution and thread assignment strategy that looks for batching queries with similar terms on the same processor. Essentially, our idea is to take advantage of the available temporal locality, to increase the cooperation between threads and avoid costly memory accesses as much as possible.

Algorithm 1 Our parallel proposal for the synchronous web search engine.

```
At the broker machine do:
// group together queries with similar terms
query_batch = build_clustered_batch(current_queries);
broadcast_to_all_processors(query_batch);

In each processor do:

#pragma omp parallel for private(...) shared(...)
for q=1:Nqueries_processor do

  query = query_batch[q];
  for term=1:terms_in_query do
    posting_list[term] = fetch(term);
  end for
  best_docs[query] = ranking(posting_list, terms_in_query);

end for
...
```

Algorithm 1 shows the pseudo-code of our parallel approach with a first step done by the *broker* machine (*master*), which tries to gather queries with some terms in common, and then each processor (*slaves*) finds the best documents associated to them in parallel.

4.2 Asynchronous mode

For the Asynchronous mode we use an *OpenMP* parallelization of the document ranking routine executed by each query to get the top K documents. This parallelization involves deploying a team of *OpenMP* threads at various points of the routine. In particular, for the cases of identical operations performed over the complete piece of posting list for each query term, we do it in parallel by letting each thread to work on a different segment of the posting list. The filtering technique is a bit more involved. It needs synchronization to update the current score barrier. Further documents down the posting list must beat this barrier in order to be considered as candidates to be included in the top K results. The

barrier must be updated concurrently by the threads. This is solved by using a critical section in the points at which this barrier is updated; though this occurs less frequently during the processing of the posting lists.

Algorithm 2 Our parallel proposal for the asynchronous web search engine.

```
At the broker machine do:
// dispatch queries when they arrive
dispatch_to_processor_P(query);

In each P processor do:

for term=1:terms_in_query do
   #pragma omp parallel private(...) shared(...)
   posting_list[term] = parallel_fetch_and_operations(term);
end for
best_docs = parallel_ranking(posting_list, terms_in_query);
...
```

5 Performance Results

In this Section we attempt to answer those questions raised above. Performance results have been obtained on a single UltraSPARC-T1 processor. This study can be viewed as a first approach of a more complex distributed system based on CMP processors which should behave as *slaves* in our context. Our objective is to analyze in detail the best intra-node parallel approaches and outline some preliminary conclusions which could be extrapolated for a real infrastructure.

5.1 Synchronous mode

Figure 2 shows the throughput achieved with our synchronous proposal. As mentioned above, we have tried to improve implicit cooperation between threads with a data distribution and thread assignment strategy that looks for batching together queries with similar terms. To estimate the potential benefits of this strategy we have compared two extreme scenarios. The gray column corresponds to the most adverse situation: there is no common terms between subsequent queries and all the threads of a given batch compete for the available resources. The black column, in contrast, corresponds to the potentially most favorable situation: all the threads of a given batch process queries with identical terms.

The noticeable difference between both scenarios when running 16 and 32 threads highlights the benefits of a conscious thread distribution. In any case, the throughput is satisfactory enough in both scenarios. The speedup increases proportionally with the number of threads and in the most favorable scenario, our synchronous search engine reaches a speedup of 22 using 32 threads.

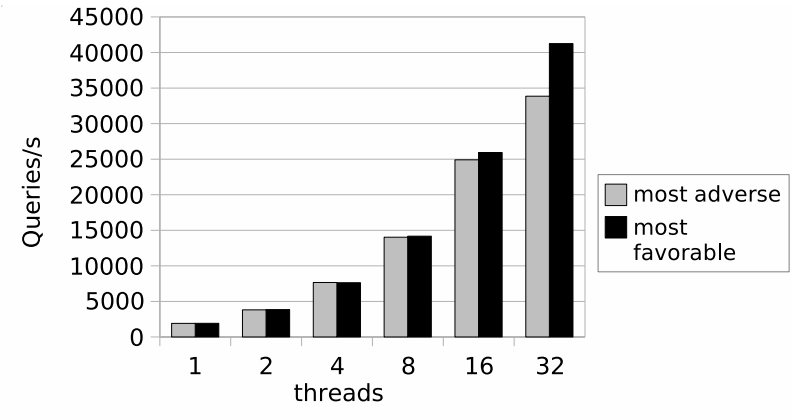

Fig. 2. Query throughput achieved by the synchronous mode in the most adverse (gray column) and favorable (black column) scenarios.

5.2 Asynchronous mode

Essentially, the experimental results (see Figure 3) show that the gain coming from parallelism is not really significant. This is mainly due to the fact that in each round-robin iteration of the processing of a given query, the amount of data (the ones involved in the pieces of posting lists of size K) that is processed is small. For example, at various points in the document ranking process it is necessary to sort candidate documents. However, trying to do that sorting in parallel by using *OpenMP* threads is not worthwhile since the amount of document to be sorted is not large enough.

Recall that round-robin is necessary to prevent large queries from consuming all resources in detriment to small queries. In addition, the filtering technique applied to avoid having to consider the complete posting list for each query term is based on the update of a barrier which finally stop the ranking by deciding that the remaining items in the involved posting lists are not able to include new document among the top K results. When implemented using *OpenMP* threads this barrier become a critical section of the ranking process whose serialization introduces performance degradation.

We should emphasize that current search engines are fully asynchronous and they are prone to this problem as well. In general these machines use techniques to avoid scanning the complete posting list and thus they are essentially in the same difficulties to get advantage of the capabilities provided by CMP systems. In this regard, it can be argued that under high query traffic, the execution of multiple queries would also overlap in the asynchronous mode and this overlapping would provide enough parallelism. In the following we call this ideal case Optimal-Async.

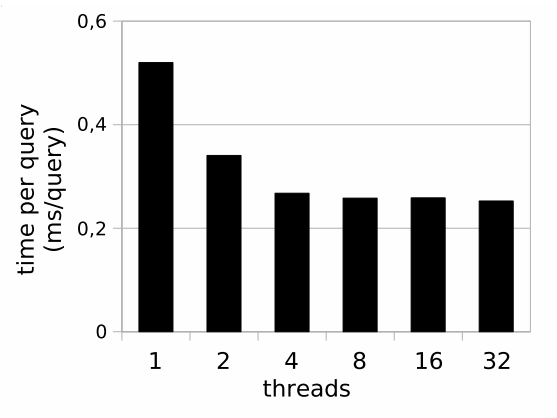

Fig. 3. Time (ms) per query using the asynchronous mode.

Figure 4 highlights that even in this case, an asynchronous engine will perform much worse than the synchronous counterpart. This figure shows the query throughput achieved by the different modes. Despite the Optimal-Async model exploits both sources of parallelism – inter and intra-query parallelism – in an idealist and optimal way (thread management has been oversimplify in these simulations). Furthermore, for a given number of threads, we report the throughput achieved by the optimal combination of inter and intra-parallelism, it does not outperform the efficient synchronous model. Essentially, the overheads caused by the asynchronous thread management and the limited intra-query parallelism introduce an upper bound to the asynchronous scalability.

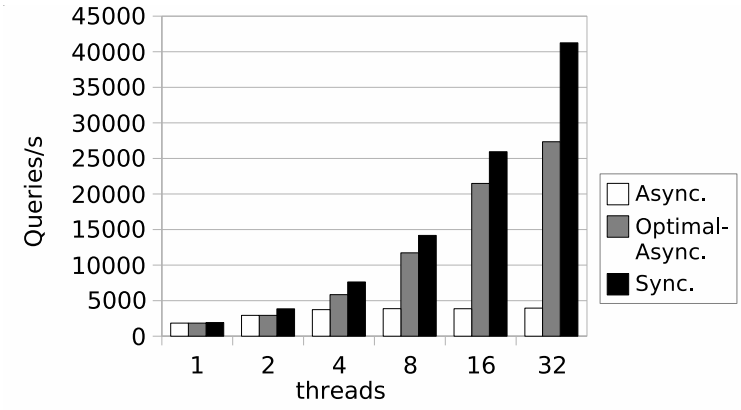

Fig. 4. Query throughput achieved by the different operation modes under study.

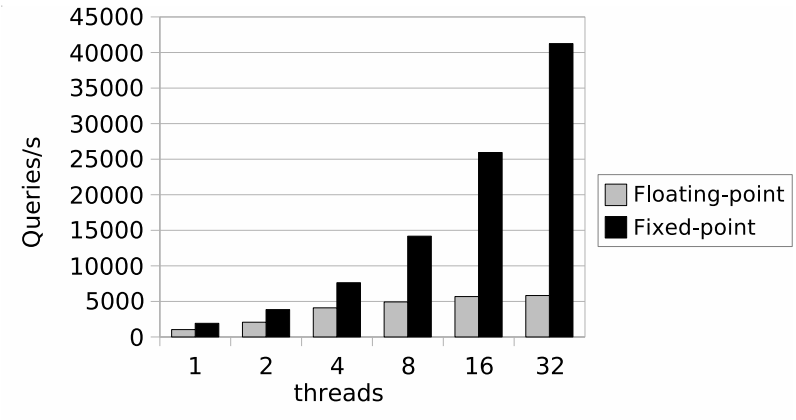

Fig. 5. Query throughput for the different numerical format representation: floating-point (gray column) and fixed-point (black column).

5.3 Fixed-point arithmetic. Impact on performance and validation

Figure 5 analyzes the impact of fixed-point arithmetic on scalability. It shows the number of queries per second that we are able to solve in the synchronous version when performing the ranking process with either floating-point or fixed point arithmetic – under high query traffic the asynchronous version behaves similarly –. As expected, the floating-point version does not scale beyond a modest number of threads.

To the best of the authors' knowledge, this is the first study that explores fixed-point arithmetic for ranking purposes. A final question here to conclude our discussion is whether the use of fixed-point operations (instead of floating-point ones) produces some effect in (1) the final set of documents selected as the answer to each query and (2) their relative position within the top K results. We evaluated this experimentally by running both fixed and floating-point document ranking functions under the same inverted file and set of queries.

We performed two tests on the set of documents generated in both cases for each query – tests on sets A for fixed-point results and B for floating-point results. The first test calculates the ratio $|A \cap B|/|B|$ for which we obtained results very close to 1; we observed average values between 0.99 and 1.0 for different and very large sets of queries. This indicates that both sets are practically identical.

The second test calculates the Pearson's correlation of the sets A and B to measure the relative position of the documents in A with respect to their position in the top K query results present in B. Again we obtained values very close to 1 indicating that there are almost no difference in the relative position of the documents A in the top K results for the queries.

6 Conclusions

A logical view of the T1 processor suggests the application of the general principles of data partitioning to get the multithreaded versions of our Web Search Engine. Essentially, this partitioning is performed running queries into separate threads.

This strategy can be easily expressed with *OpenMP* directives. However, the similarities amongst the different threads may cause contention for shared resources, especially cache space and memory bandwidth. We have tried to minimize these effects with a data distribution and thread assignment strategy that looks for batching on the same processor and tries group together queries containing common terms. This strategy aims at taking advantage of the temporal data locality, and to avoid the costly memory access.

As further research we plan to increase locality by devising strategies that reorganize the way in which the chunks of size K of posting are stored in main and secondary memory in order to exploit locality. This should be made by taking into consideration how frequently the terms appears in queries together, which can be obtained from the logs that search engines maintain at their data centers.

References

1. A. Arusu, J. Cho, H. Garcia-Molina, A. Paepcke, and S. Raghavan. Searching the web. *ACM Trans.*, 1(1):2–43, 2001.
2. K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K.A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.
3. 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 (SPIRE01)*, pages 10–20, 2001.
4. A. Barroso, J. Dean, and U. H. Olzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23(2):2002, 22-28.
5. J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.
6. P. Kongetira, K. Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, 2005.
7. M. Marin, C. Bonacic, V. Gil-Costa, and C. Gomez. A search engine accepting on-line updates. In *Euro-Par '07: 13th International Conference on Parallel and Distributed Computing*, pages 348–357, 2007. LNCS 4641.
8. M. Marin and V. Gil-Costa. High-performance distributed inverted files. In *CIKM '07: Proceedings of the Sixteenth ACM Conference on Information and Knowledge Management*, pages 935–938, New York, NY, USA, 2007. ACM.
9. M. Marin and V. Gil-Costa. (Sync|Async)$^+$ MPI Search Engines. In *PVM/MPI*, pages 117–124, 2007. LNSC 4757.

10. W. Moffat, J. Webber, J. Zobel, and R. Baeza-Yates. A pipelined architecture for distributed text query evaluation. *Information Retrieval, published on-line*, 5:2006, October.
11. K. Olukotun, L. Hammond, and J. Laudon. *Chip Multiprocessor Architecture: Techniques to Improve Throughput and Latency.* Number 3 in Synthesis Lectures on Computer Architecture. Morgan and Claypool Publishers, 2007.
12. 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.
13. D. Sheahan. Developing and tuning applications on ultrasparc t1 chip multithreading systems. Technical report, Sun Microsystems. Sun BluePrints Online, October 2007.
14. J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38(2), 2006.