Scheduling Metric-Space Queries Processing on Multi-Core Processors

Veronica Gil-Costa[1] Ricardo J. Barrientos[3] Mauricio Marin[2] Carolina Bonacic[4]
[1]*DCC, University of San Luis, Argentina*
[2]*Yahoo! Research Latin America*, [3]*DCC, University of Chile*
[4]*ArTeCS Group, Complutense University of Madrid, Spain*

***Abstract*—This paper proposes a strategy to organize metric-space query processing in multi-core search nodes as understood in the context of search engines running on clusters of computers. The strategy is applied in each search node to process all active queries visiting the node as part of their solution which, in general, for each query is computed from the contribution of each search node. When query traffic is high enough, the proposed strategy assigns one thread to each query and lets them work in a fully asynchronous manner. When query traffic is moderate or low, some threads start to idle so they are put to work on queries being processed by other threads. The strategy solves the associated synchronization problem among threads by switching query processing into a bulk-synchronous mode of operation. This simplifies the dynamic re-organization of threads and overheads are very small with the advantage that the overall work-load is evenly distributed across all threads.**

I. Introduction

Currently multi-core processors such as the Intel Xeon Quad-core or the Intel Core i7 provide 4 physical CPUs and 8 logical ones to the programmer that can be used with `C++` programs and libraries such as OpenMP [1]. This software combination allows one to have P threads running in parallel, each one in a different CPU, under a scheme in which all threads have access to the same main memory. The challenge is to reduce the total running time P times where for some applications, like metric-space databases [19], [21], this can be tricky to achieve because one has to reduce resource competition and prevent duplicated calculations at low overheads.

In metric-space databases the collection of objects is indexed by using data structures and search algorithms that employ as a primary tool a function that computes the distance between any two objects. Queries are objects of the same type and the index is used to quickly retrieve the objects that are most similar to a query object in terms of their distances to it. The distance function is expensive to compute in running time so the main objective of the index is to reduce the number of distance evaluations among objects.

This paper proposes an efficient strategy to solve range queries upon different metric space indexes by using P threads in the shared memory model of multi-core processors. The strategy optimizes the load balance of distance evaluations among database objects performed by the P threads executing in parallel, where good load balance is achieved at reduced competition for shared data and efficient use of all threads.

A. Problem context

Metric-space data structures allow threads to interrupt at any point the sequence of distance evaluations among objects that are necessary to completely solve a given query. This is easily achieved by keeping a small section of memory with state information that allows a thread to continue with the next distance evaluation during query processing. Given its high cost, each distance evaluation operation can be considered as a unit of work so that the multi-core query processing strategy can only focus on properly scheduling those units onto the concurrent threads to achieve efficient performance.

Another key feature of metric-spaces is that the execution of a given unit of work can generate a large number of new units of works that are independent each other in terms of data dependencies and thereby can be executed in parallel. The problem is to detect such cases without incurring in overheads coming from excessive synchronization and look-ups by ways of lock operations that introduce serialization.

When a search node with P CPUs has less than P active queries, the desired aim is to let idling threads help busy threads by allowing them to steal units of work from the busy ones. The number of active queries can vary dynamically in a search node because of at least two reasons: (a) There are no new queries in the node input message queue at a given time interval, and (b) some queries are momentarily blocked waiting for secondary memory operations to be completed. On the other hand, as soon as there are P active queries being solved the ideal case is to just let each thread work entirely on the processing of a single query. This is the optimal case because no synchronization among threads is necessary since all accesses to the data structure are read-only.

B. The proposed solution

This paper proposes an efficient strategy to schedule the above units of work to be processed in parallel by P threads. When there are P or more active queries in the search node our strategy processes them by using naive parallelism, namely each thread completely processes a single query and then fetches the next one from the input message queue and so on. This represents a situation in which the query traffic is high. As soon as the level of active queries decreases below P we put the idling threads to work collaboratively in the processing of the existing queries being solved. Here, therefore, we have the above mentioned resource contention situation which requires synchronization of threads in order to re-schedule the units of work.

We solve the contention problem by dynamically switching the computation into a bulk-synchronous one involving all P threads. In this case the threads work in an asynchronous manner for a while by processing a certain number N_w of units of work. In the process, each thread stores in a local queue the new units of work that it generates and must be processed by other threads. To ensure good load balance, a round-robin rule is used to evenly distribute the units of work onto the P threads.

At some point the limit N_w for the total number of units of work allowed to be processed by all threads is reached and they are all barrier synchronized. The distance evaluation calculations are stopped no matter in which sections of the data structure the threads are at that instant (for instance if a thread is traversing the node of a tree, the remaining distance evaluations required to complete the processing of the node are delayed until the above asynchronous step is repeated).

After the synchronization point all threads start to scan in a read-only manner the local queues of all other threads looking for units of work that are scheduled for them (in a classical solution based on fully asynchronous parallelism and locks this would require locking queues to get or store units of work which serializes the computation). This scanning process is performed in parallel by all threads. Once the threads have obtained their new units of work a last barrier synchronization is performed and the first asynchronous step is repeated.

We refer to the above two fully asynchronous steps delimited by the barrier synchronization as *supersteps* since they resemble the BSP model of parallel computing [20].

The reasons that explain the efficient performance of the proposed scheduling algorithm are that (a) the distance evaluations performed in the first superstep are perfectly balanced across the P threads, (b) the cumulative granularity (running time cost) of them is large in comparison with the computations involved in the second superstep and the cost of the synchronization barriers, and (c) the scan phase in the second superstep is performed in perfect parallelism which further reduces its cost and no read/write conflicts ever take place since each thread reads other queues and writes its own local queue to store its new units of work. The trade-off – represented by the number of units of work allowed to be processed in the first superstep – is determined experimentally.

Once the query traffic is detected to be high enough, the thread computations are switched to the fully asynchronous mode where each thread processes a single query completely before starting with a new one. This situation can be easily detected by testing the length of the input query queue of the node. Detecting the opposite situation of low query traffic is also simple since threads start to idle.

C. The big picture

In a production system, the overall number of search nodes can be set in such a way that at normal query traffic there are $p < P$ active queries in each search node at any time, so that the fully asynchronous mode with one thread per query is triggered upon sudden increments in the query traffic.

In addition, each search node can contain a metric-space cache [12] which is used to prevent frequent queries from being recomputed. Thus upon the arrival of a new query to the search node three steps take place: (1) a search on the cache is performed in order to detect whether an answer for the query is already there, (2) if not, the query is solved concurrently (which is the subject of this paper), and (3) once the processing of the query is completed, the results are stored in the cache using an eviction policy such as LRU (in a multi-threaded system this produces a concurrency control issue). Solving the step 3 can be demanding in running time since distance evaluations must be performed in order to keep properly indexed the queries stored in the cache memory. Certainly this task is less demanding than solving the query against the large index that indexes all of the database objects stored in the search node. Nevertheless, R/W concurrency control on the cache increases overheads.

The issue of concurrent updates on the search node cache is out of scope in this paper. In a previous work we show that the bulk-synchronous mode of computing is particularly suitable for the task of updating the cache index when chunks of queries are available for insertion [16]. A central part of cache management is the parallel priority queue used to determine what entries must be evicted from the cache. This can be implemented by using a multi-core parallel priority queue as the one proposed in [16].

The point of this paper is to show that similar mode of multi-threaded processing is also suitable for the step 2 above in the sense that we use this mode to (a) generate a new chunk of queries from the currently active queries in the search node and (b) processes this chunk in bulk during step 3. This paper shows that idle threads can dynamically be assigned to help query processing during step 2 and that their inclusion effectively reduces running time and produces a new chunk of queries very quickly. For periods of sudden peaks in query traffic, the step 3 is not executed and the cache update is delayed until traffic is restored to normal, to then include the results from the queries solved during the peak period.

The remaining of the paper is organized as follows. Section II introduces the metric space concepts and presents related works. Section III presents the proposed Local, Bulk-Circular and Bulk-Local multi-core strategies. Section IV shows how to apply the multi-core scheduling strategies over different metric space indexes. Section V shows the databases used in the experiments and results obtained by all metric space indexes. Section V also shows how to combine the Local and Bulk-Circular strategies to obtain the proposed multi-core query processing scheduler. Finally Section VI presents conclusions.

II. Related Work

Searching sequentially for all objects which are similar to a given query object is a problem that has been widely studied in recent years. A typical query for these applications is the *range query* which consists on retrieving all objects within a certain distance from a given query object. That is, finding a set of *similar* objects to a given object. The solutions are based on the use of a data structure that acts as an index to speed up the processing of queries. Applications are diverse such as voice and image recognition, and data mining.

Similarity can be modeled as a metric space as stated by the following definitions.

Metric Space. A *metric space* $(\mathbb{X}, d)$ is composed of a universe of valid objects $\mathbb{X}$ and a *distance function* $d : \mathbb{X} \times \mathbb{X} \rightarrow \mathbb{R}^+$ defined among them. The distance function determines the similarity between two given objects. This function holds several properties: strictly positiveness ($d(x, y) > 0$ and if $d(x, y) = 0$ then $x = y$), symmetry ($d(x, y) = d(y, x)$), and the triangle inequality ($d(x, z) \leq d(x, y) + d(y, z)$). The finite subset $\mathbb{U} \subset \mathbb{X}$ with size $n = |\mathbb{U}|$, is called collection or database and represents the set of objects where searches are performed.

Range query. Given a metric space $(\mathbb{X}, d)$, a finite set $\mathbb{U} \subset \mathbb{X}$, a query $x \in \mathbb{X}$, and a range $r \in \mathbb{R}$. The results for query x with range r is the set $y \in \mathbb{U}$, such that $d(x, y) \leq r$.

The k nearest neighbors (k-NN). Given a metric space $(\mathbb{X}, d)$, a finite set $\mathbb{U} \subset \mathbb{X}$, a query $x \in \mathbb{X}$ and $k > 0$. The k nearest neighbors of x is the set A in $\mathbb{U}$ where $|A| = k$ and $A = \{u | d(u, x) < d(y, x)\} \forall y \in \mathbb{U} - A$.

In this work we focus on range queries because k-NN queries can be efficiently solved using range queries [7]. The distance between two database objects in a high-dimensional space can be very expensive to compute and in many cases it is certainly the only relevant performance metric to optimize (they are even more expensive than the cost of secondary memory operations). Thus for large and complex databases it becomes crucial to reduce the number of distance calculations in order to achieve reasonable running times.

Search methods can be classified in two groups [8]: pivot-based and clustering-based search methods. Pivot-based methods select a subset of objects from the collection as pivots, and the index is built up by computing and storing the distances from each pivot to the objects of the database. During the processing of a query, the pre-computed distances are used to test the triangle inequality to discard objects that in other case would have been compared against the query. Comparing two objects involves calculating the distance between them. The objects that the triangle inequality is not able to discard are compared against the query. Clustering-based methods partition the metric space in a set of zones or clusters, each of them represented by a cluster center. During query processing, complete regions are discarded based on the distance from their centers to the query so that the objects belonging to those regions are not compared against the query.

There are a number of programming libraries for multi-core systems like TBB [18] which uses parallel loops to describe data parallelism, others like IBM X10 [6] and Fortress [3] which focus on data parallelism but task parallelism is also supported. In this work we used OpenMP as the programming library because it has a high level of abstraction and we have found it to be very efficient. An important advantage is that resulting codes are not very different from the sequential ones which is convenient for maintenance purposes. This because we do not use locks and our only synchronization primitive are barriers placed at a few points in the program. Another good candidate to implement our approach is PThreads but in this case the code becomes quite more populated with library calls.

There are a number of methods proposed to schedule tasks over a set of cores. For instance, [14] shows how to reduce task pool overheads, [2] proposes a scheduler using information to determine the number of processors assigned to execute a job. [11] allows stealing tasks from a queue using a specific data structure. We did not find any related work in the literature for multi-core systems applied to problems with the features of metric space indexes where computing results for a similarity search query requires sharing a great amount of intermediate results and where some tasks generate an unpredictable number of new tasks. In the introductory section of the paper we called these tasks as units of work, namely each task involves executing a distance calculation between two objects.

III. Multi-core scheduling strategies

In this section we explain the details of the proposed multi-core scheduling strategies assuming that a search node receives queries from outside and place them in a lock-protected input queue IQ which is shared by all threads. Each thread is assumed to be executed in a different CPU. The metric space index is stored in the main shared memory. During search node operation, each thread takes a query from the input queue IQ and processes it by using one of the following algorithms:

- **Local**: In this strategy neither data sharing nor periodical synchronizations are required because each thread completely processes a query by using the sequential algorithm. When a thread becomes idle, namely it has found all the answers for the query, it locks the input queue IQ up to remove the next query from IQ and process it.
- **Bulk-Circular**: We define a *task requirement* as a piece of data that contains information of the specific job assigned to a thread such as the next node of the index to be examined. A task requirement usually involves calculating the distance between objects and the (much less costly) application of the triangle inequality. Each time the algorithm processes a query it may generate a set of task requirements that are stored in special purposes queues. To this end, each thread has a private local requirement queue Q_{PR} and a secondary requirement queue Q_{SR} that maintain task requirements to be solved in the next supersteps. A superstep is a sequence of tasks executed in parallel by threads and delimited by the barrier synchronization of all of them. In a given superstep, Q_{SR} is written by the owner thread and read by the other threads in the next superstep, so there is no read/write conflict.

  The processing of all active queries takes place in pairs of supersteps that are repeated whilst the search-node is operating in this bulk-synchronous mode. In the first superstep, all threads execute the task requirements stored in their Q_{PR} queues and place new requirements in their secondary queues Q_{SR}. If a Q_{PR} is empty, the respective thread checks whether there is a query waiting for service in the search-node input queue IQ and inserts it in its private local queue Q_{PR}. When a thread generates a new task requirement to be placed in Q_{SR}, it assigns a thread to process it by selecting the thread with the least amount of assigned task requirements as indicated from the previous pair of supersteps and its current count. We also limit the number of distance evaluations performed by each thread to N_w per superstep. The aim is to properly load balance the computations performed by all CPUs. The value of N_w is adjusted experimentally for each index and database. During the second superstep, the threads copy from all other thread queues Q_{SR} the task requirements assigned to them into their local queues Q_{PR}. A pseudo-code describing this strategy is presented in ALGORITHM 1.

- **Bulk-Local:** Similar to the Bulk-Circular strategy but the new requirements generated in the first superstep are assigned to the thread that originally took the query from the search-node input queue IQ.

Our proposal is using *Local* for high query traffic and *Bulk-Circular* for moderate to low query traffic and a method for automatically selecting one of them from the observed query traffic. The Bulk-Local strategy is included as an intermediate case for comparison purposes. Also for comparison, though its comparative performance is very bad, we enhance the Local strategy with a lock-based strategy in which idle threads take units of work from threads that are currently processing queries (we provide the details of this fairly standard approach in the experiments section).

Algorithm 1 Searching using the Bulk-Circular Strategy.

```
ThreadQueryProcessing( pid )
1:  while bulkMode = true do
2:    while limit < $N_w$ do
3:      if $Q_{PR}$.empty() = true then
4:        task ← nextQuery( $IQ$ );
5:        $Q_{PR}$.insert( task );
6:      end if
7:      task ← nextTask( $Q_{PR}$ );
8:      taskList ← executeTask( task );
9:      for each task in taskList do
10:       if task.targetThread = pid then
11:         $Q_{PR}$.insert( task );
12:       else
13:         $Q_{SR}$[pid].insert( task );
14:       end if
15:     end for
16:   end while
17:   #pragma omp barrier
18:   for $i$=0; $i < P$; $i$++ do
19:     if $i$ != pid then
20:       for $j$=0; $j < Q_{SR}[i]$.size(); $j$++ do
21:         if $Q_{SR}[i][j]$.targetThread = pid then
22:           $Q_{PR}$.insert( $Q_{SR}[i][j]$ )
23:         end if
24:       end for
25:     end if
26:   end for
27:   #pragma omp barrier // Threads synchronization.
28:   $Q_{SR}$[pid].clear();
29: end while
```

IV. ADAPTING METRIC-SPACE INDEXES

We have selected five different metric space indexes to test our multi-core strategies: the EGnat [15], SSS-tree [5], LC-SSS [17], SSS-Index [4] and M-Tree [9]. For the Local multi-core strategy we have applied the original sequential algorithms for each index and for the bulk strategies we have adapted the algorithms to let them work as sequences of tasks.

The SSS-Index is a table with pivots in the columns and objects in the rows. Each cell stores the distance $d(p_i, o_j)$ between the pivot p_i and the object o_j. Pivots are selected using the SSS [4] strategy. When we apply the bulk strategies to the SSS-Index the receptionist thread (selected in a circular way) takes a query q from the IQ queue and computes the distance between the query object and all of the pivots. Then using the triangle inequality it selects a set of candidate objects that must be compared against the query. For each object in the candidate set, the receptionist thread generates a task requirement and place it in the Q_{SR} queue. These task requirements are assigned to the thread with the least load when the Bulk-Circular strategy is being used. When using the Bulk-Local strategy, these task requirements are assigned to the receptionist thread.

The LC-SSS index is composed of a set of clusters where each one i contains a center c_i, a covering radius r_i and K objects from the database. The value of K is experimentally determined to find the value that produces the best performance. Inside each cluster we set a table of pivots where the pivots are selected using the SSS strategy [4]. During query processing, a receptionist thread (selected in a circular way for different queries) takes a new query q from the IQ queue and computes the query plan [10] which means determining the clusters that intersect the query ball (q, r). The processing of each of these clusters is assigned in a circular manner among the threads. Processing a LC-SSS cluster involves using the pivots to determine a set of objects that must be compared against the query. The respective distance evaluations are also considered task requirements that must be assigned to the threads for processing. If the Bulk-Circular strategy is being used, these task requirements are assigned to different threads circularly and placed in the local Q_{SR} queue by the respective thread. If the Bulk-Local strategy is being used, the task requirements are assigned to the receptionist thread.

The EGnat uses m cluster centers in each node of a m-ary tree. Each node of the tree stores a table with m rows (one for each cluster center in the node) and 2 columns. Cell $(i, 1)$ stores the minimum distance and cell $(i, 2)$ stores the maximum distance from the first cluster center in the node to any object stored in cluster i. These values are used to apply the triangular inequality to discard tree branches containing no objects in the query results. The receptionist thread takes a query from IQ and compares it with the first object of the root node. Then it uses the table to select the candidate branches of the tree where to continue the search by using the triangle inequality. For each candidate branch we generate a task requirement which is assigned following the same strategy of the above described indexes.

The SSS-Tree is similar to the EGnat but the number of objects stored in each node are determined by the SSS strategy. In this case the objects of internal nodes are centers of clusters c_i, and we go down through a branch when $d(q, c_i) - r \geq rc(c_i)$, where the covering radius $rc(c_i)$ is the distance from the cluster center c_i and the farthest object in the cluster.

The M-tree is built in a bottom-up way and it is composed by nodes with at most m objects. When a node is full the insertion algorithm splits it into two new nodes and generates a new father node. The query processing is similar to the EGnat. Than it, the receptionist thread computes $d(q, o_r)$ for all objects o_r in the root node and obtains the candidate branches to go down during search. For the internal nodes, the triangle inequality is used to discard branches. For each candidate branch to be visited, we have a new task requirement stored in the Q_{SR} queue. These task requirements are processed by different threads when the Bulk-Circular strategy is being used or by the receptionist thread when using the Bulk-Local strategy.

V. Experimental results

The results were obtained with two different databases containing objects from two types of metric spaces. We took a collection of images from a NASA database containing 40,700 images vectors, and we used them as an empirical probability distribution from which we generated a large collection of random image objects containing 200,000 objects. We built each index with the 80% of the objects and the remaining 20% objects were used as queries. In this collection we used the Euclidean distance to measure the similarity between two objects with radii 0.47, 0.57 and 0.73. This allowed the query processing algorithms to retrieve, on the average, 0.01%, 0.1% and 1% of the database objects respectively. The second database is a Spanish dictionary with 51,589 words and we used the Edit distance to measure similarity with radii 1, 2 and 3. On this metric-space we processed 40,000 queries selected from a sample of the Chilean Web which was taken from the TODOCL search engine.

The experiments were performed on a machine composed by two Intel's Quad-Xeon (2.66 GHz) multi-core processors with 16 GB of RAM. The thread scheduler provided by the *sched.h* library allowed us to assign each of the 8 threads to a different CPU. We had exclusive access to these cores. In the following we show results normalized to 1 in order to better illustrate the comparative performance among the scheduling strategies using 8 CPUs.

Figure 1 shows the performance of the three multi-core scheduling strategies for all indexes using the Spanish data collection (at the top), and the NASA image collection (at the bottom). In this experiment all queries have arrived at the same instant to the system, simulating high query traffic. The LC-SSS and SSS-Index tends to report similar results for all multi-cores strategies with a low query radius because these indexes reduce significantly the number of objects compared against the query. In all cases, the Local strategy reports the best performance because under a high query traffic all threads are continuously processing distance evaluations in parallel and in an asynchronous manner, whereas the bulk strategies have the additional cost of synchronization and task assignment to threads.

In Figure 2 queries arrive at different time intervals so some threads may get none or very little work to do. Unlike the results reported in Figure 1, the results obtained by all indexes in this experiment show that the Bulk-Circular strategy achieves the best performance. This is because threads that become idle help others to solve their queries.

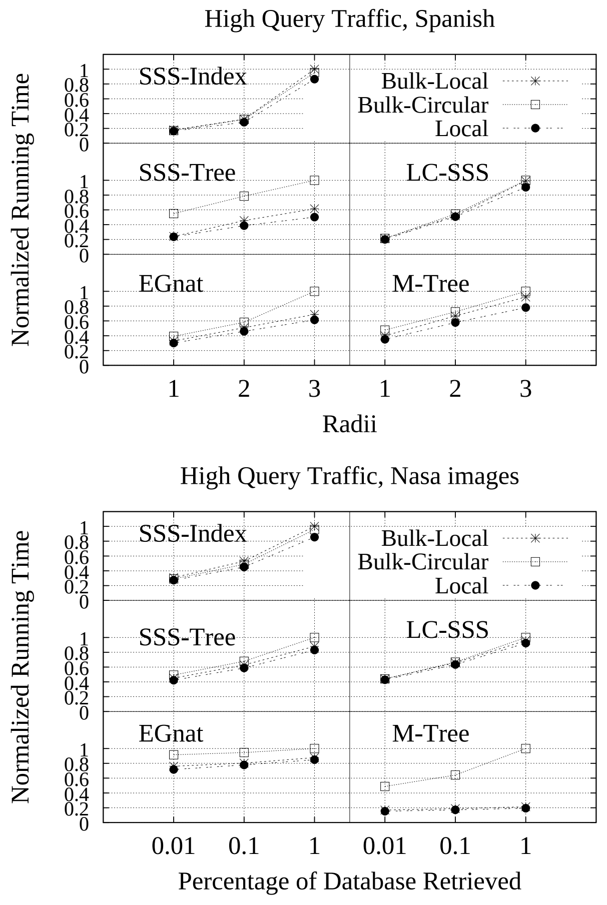
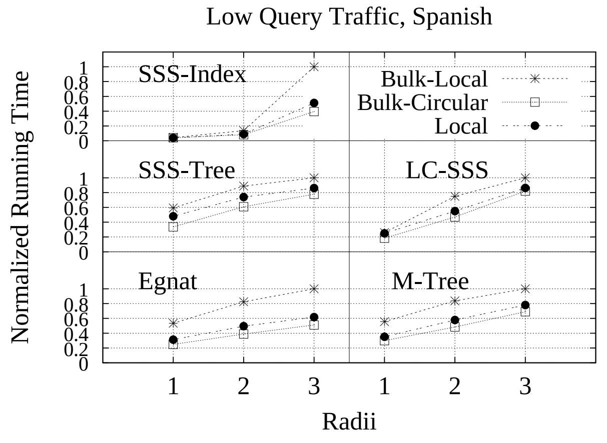


Fig. 1. Performance obtained by all indexes under high query traffic.

Figure 3 compares the performance of all indexes under high query traffic for the Spanish and NASA images databases using the Local scheduling strategy which reported the best performance for this case. The LC-SSS index achieves the best performance with radii 1 and 2 using the Spanish collection, but the SSS-Tree is better for radius 3. Using the NASA collection we have similar results for the M-tree, LC-SSS, and SSS-Tree with small radii. With the third radius the M-Tree achieves the best performance.

Figure 4 compares the performance of all indexes under a



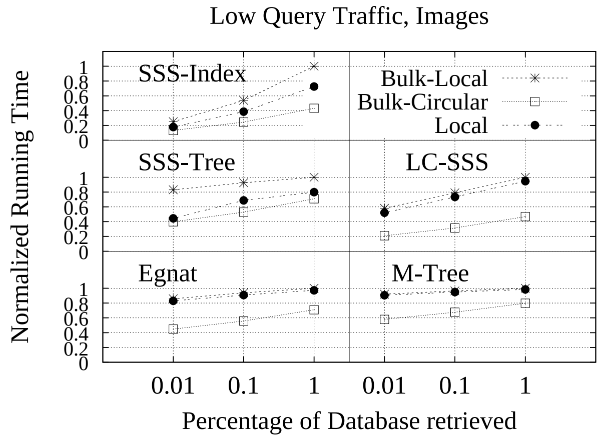
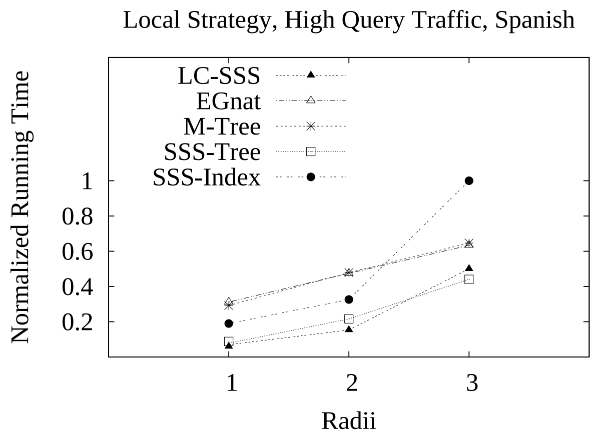


Fig. 2. Performance obtained by all indexes under low query traffic.



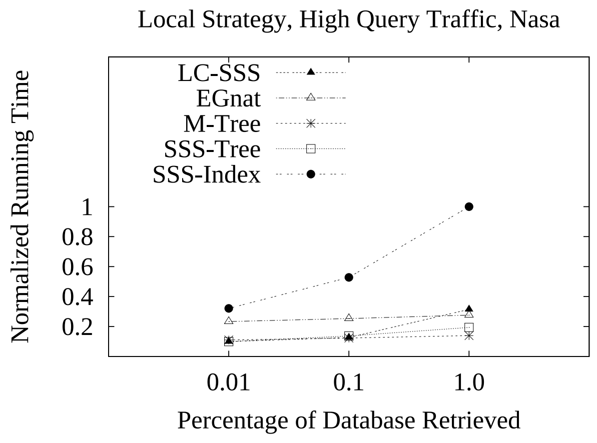


Fig. 3. Performance of all indexes under high query traffic.

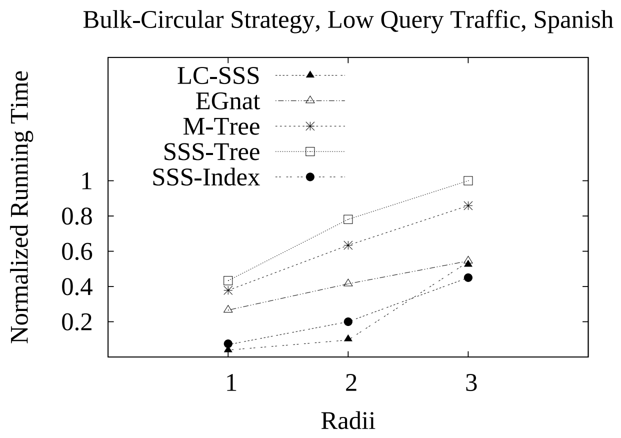
situation of low query traffic using the Bulk-Circular strategy. This strategy achieves the best performance for this query traffic. For the Spanish collection the LC-SSS tends to achieve better performance than the other indexes with radii 1 and 2 because it combines two indexes to reduce the number of distance evaluations performed per query. For the NASA collection the LC-SSS and SSS-Index present similar results.

Figure 5 shows the speed-up achieved by all indexes running over the Spanish database with radius 1. The results show speed-ups achieved by all three scheduling strategies. For high query traffic, Figure 5 [top], the results show that all indexes achieve the highest speed-ups with the Local scheduling strategy. On the other hand, the Figure 5 [bottom] shows that the Bulk-Circular strategy achieves the highest speed-ups for all indexes when the query traffic is low.

Figure 6 shows the efficiency obtained using the LC-SSS index with the Local, Bulk-Local and Bulk-Circular scheduling strategies. The efficiency is measured as $(\sum w_i/max_w)/P$ where w_i is the work-load in each thread i, max_w is the maximum work-load detected in any of the threads and P is the number of threads. A value close to 1 indicates a well load balanced system. These results were obtained under low query traffic and the Bulk-Circular strategy achieves the best efficiency.

To understand the results of Figure 6, in Figure 7 we show the effect of using the Local and Bulk-Circular scheduling strategies under low query traffic. In this example we assume that each thread computes only one distance evaluation per unit of time. At time t_1 two queries q_1 and q_2 arrive to the system. q_1 requires four distance evaluations (DEs) and q_2 requires three DEs. q_3 arrives at time t_2 requiring two DEs, then q_4 arrives at t_4 requiring one DE. This sequence is repeated for 12 queries (meaning at t_5 two new queries arrive requiring 4 and 3 DEs each, at t_6 one query arrives requiring 2 DEs and so on). The left table shows the work-load assigned to each thread when queries are assigned in a circular manner when using the Local strategy. The right table shows the work-load when the queries are assigned to the least load thread. In both cases we can see the effect produced by the Local scheduling strategy and how the queries are delayed reducing the throughput and increasing the query response time for low query traffic.

Figure 8 shows the performance of the LC-SSS using the scheduling strategies Local and Bulk-Circular, and compares them with a Lock based strategy. In the Lock strategy, each query is processed as in the Local strategy but when a thread becomes idle, it search into the other thread's Q_{SR} queues for new task requirements and place them in its local queue Q_{PR}. In order to steal a task requirement, the thread selects the thread with the largest number of task requirements, and then it steals half of them. This strategy uses locks to prevent reads/write conflicts over the threads queues. The results shows that in a low query traffic situation, the Bulk-Circular strategy achieves the best performance. The Lock strategy incurs in a high overhead caused by the locks mechanism. When the



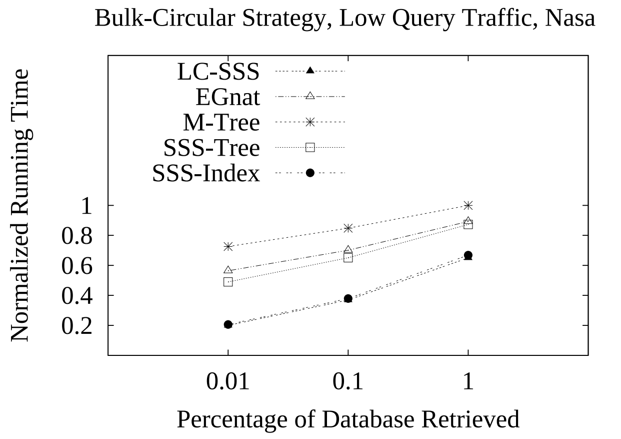
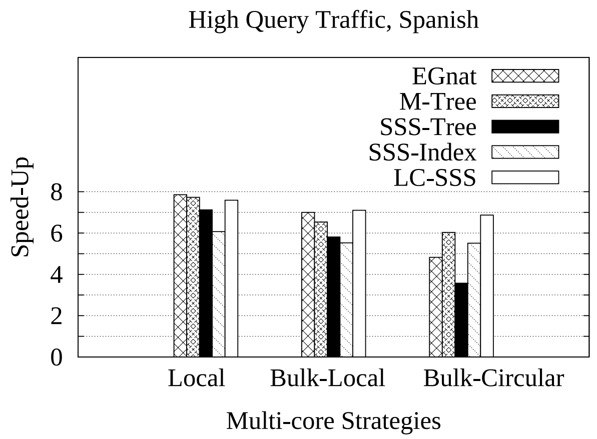


Fig. 4. Performance of all indexes under low query traffic.



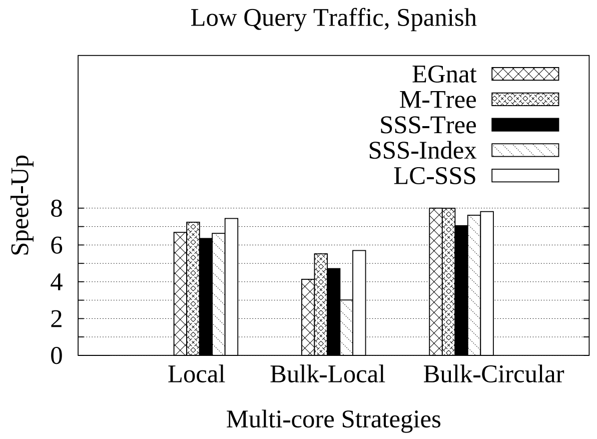


Fig. 5. Speed-Up obtained by all indexes with radius 1, under a high query traffic [top] and low query traffic [bottom].

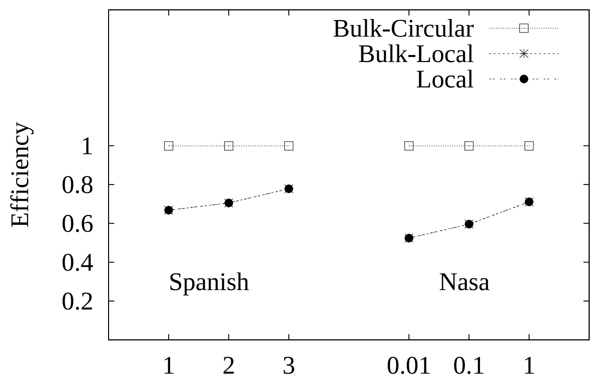


Fig. 6. Efficiency achieved by the Local, Bulk-Circular and Bulk-Local scheduling strategies using the LC-SSS index under low query traffic.

query traffic is high and each thread is continuously solving different queries, the Lock strategy achieves the same performance than the Local strategy. This is because no stealing operations are performed and all threads are always busy.

A. Switching between strategies

The Bulk-Circular scheduling strategy achieves good performance when the query traffic is low because the number of distance evaluations required by a query are distributed among all threads keeping them busy all the time. On the other hand, when the query traffic is high it is more convenient to assign a unique query to a single thread avoiding sharing local data. Therefore we have implemented a hybrid scheduling strategy that is able to change from one type of query processing to the another by measuring the number of queries requiring service.

Figure 9 compares the throughput obtained by the Bulk-Circular, Local and the hybrid scheduling strategies when the query traffic changes along time. The results show that the hybrid strategy is able to set itself in the mode that achieves the best performance.

When the number of queries Q_p waiting to be completed in the processor satisfies $Q_p \geq P * Q_{max}$, where P is the number of threads and Q_{max} is the maximum number of queries allowed to be processes in the bulk-synchronous mode, the hybrid strategy changes its operation mode to the Local scheduling strategy. After a time, when the number of tasks assigned to each thread is low enough, the hybrid strategy changes to the Bulk-Circular operation mode. Figure 9 shows that the hybrid strategy achieves the highest query throughput especially when the query traffic changes from moderate to high.

VI. Conclusions

In this paper we have proposed a scheduling strategy designed to improve query throughput in a multi-core server for metric-space databases. Our proposal combines fully asynchronous multi-threaded processing of queries with bulk-synchronous multi-threaded query processing in a shared

Circular distribution of queries

Time	Local				Bulk−Circular			
	T1	T2	T3	T4	T1	T2	T3	T4
1	q1	q2			q1	q1	q1	q1
2	q1	q2	q3		q2	q2	q2	q3
3	q1	q2	q3	q4	q3	q4		
4	q1	q6			q5	q5	q5	q5
5	q5	q6	q7		q6	q6	q6	q7
6	q5	q6	q7	q8	q7	q8		
7	q5	q10			q9	q9	q9	q9
8	q5	q10	q11		q10	q10	q10	q11
9	q9	q10	q11	q12	q11	q12		
10	q9							
11	q9							
12	q9							

Least loaded thread selection for queries

Time	Local				Bulk−Circular			
	T1	T2	T3	T4	T1	T2	T3	T4
1	q1	q2			q1	q1	q1	q1
2	q1	q2	q3		q2	q2	q2	q3
3	q1		q3	q4	q3	q4		
4	q1	q5	q6		q5	q5	q5	q5
5		q5	q6	q7	q6	q6	q6	q7
6	q8	q5	q6	q7	q7	q8		
7	q9	q5	q10		q9	q9	q9	q9
8	q9		q10	q11	q10	q10	q10	q11
9	q9	q12	q10	q11	q11	q12		
10	q9							

Fig. 7. Work load assignments for the Local and Bulk-Circular scheduling strategies under low query traffic.

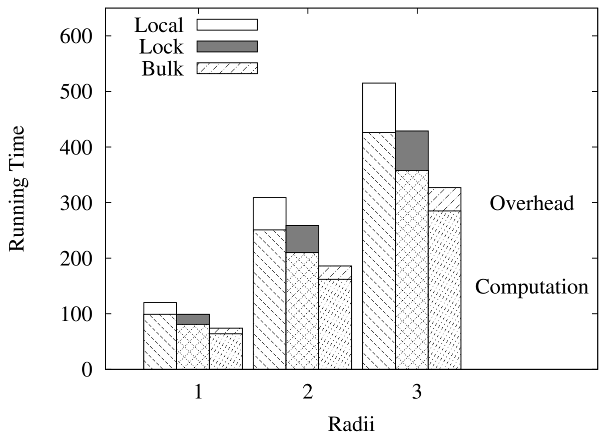


Fig. 8. Running time obtained by the Lock, Local and Bulk-Circular scheduling strategies under low query traffic.

memory setting. The switching between the two modes is made by a simple rule which acts in accordance with the observed query traffic. The comprehensive experimental study presented in the paper, which includes data structures of several types, shows that the proposed scheme is efficient in practice and it is independent of the particular index data structure. Few modifications to the query processing regime upon these data structures are necessary to make the scheduling strategy work efficiently on them. As explained in the introductory section of the paper, the bulk-synchronous scheduling strategy is devised to work in tandem with the cache update phase at each multi-core search node. We refer to an application cache in charge of keeping in main memory the answers to most frequent user queries. Previous work [16] has shown that it is more efficient to update the cache by using chunks of queries rather than single concurrent queries in order to reduce overheads and the proposed scheduling strategy does exactly that, namely it produces chunks of queries in an efficient manner which can be then cached also efficiently.

REFERENCES

[1] The OpenMP API specification for parallel programming. http://openmp.org

[2] K. Agrawal and Y. He and E. Leiserson. Adaptive work stealing with parallelism feedback. In Principles and Practice of Paralle Computing, pages 112-120. 2007.

[3] E. Allen and D. Chase and J. Hallett and V. Luchangco and W. Maessen and S. Ryu and S. Tobin-Hochstadt, S. The Fortress Language Specification, version 1.0beta. 2007.

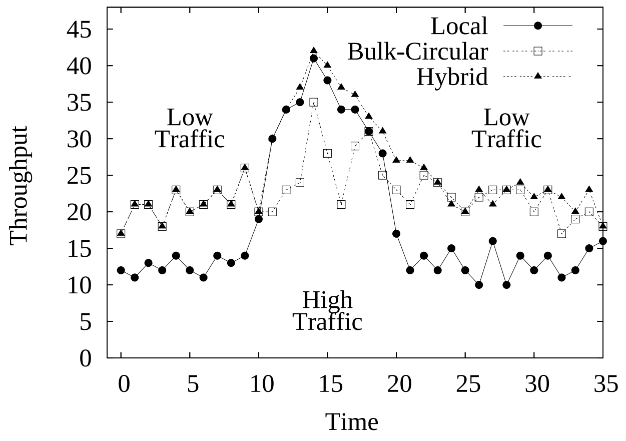


Fig. 9. Throughput obtained by all scheduling strategies under different query traffics.

[4] N. Brisaboa, A. Farina, O. Pedreira, and N. Reyes. Similarity search using sparse pivots for efficient multimedia information retrieval. In Proc. of the Eighth IEEE International Symposium on Multimedia, pages 881-888, Washington, DC, USA, 2006. IEEE Computer Society.

[5] N. R. Brisaboa, O. Pedreira, D. Seco, R. Solar, and R. Uribe. Clustering-based similarity search in metric spaces with sparse spatial centers. In Proc. SOFSEM, LNCS, 2008.

[6] P. Charles and C. Grothoff and V.A. Saraswat and C. Donawa and A. Kielstra and K. Ebcioglu and von Praun and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In Proc. of OOPSLA. pages 519538. 2005.

[7] E. Chavez and G. Navarro. A compact space decomposition for effective metric indexing. Pattern Recognition Letters, 26(9):1363-1376, 2005.

[8] E. Chvez, G. Navarro, R. Baeza-Yates, and J. L. Marroqun. Proximity searching in metric spaces. ACM Computing Surveys, 33(3):273321, 2001.

[9] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An effcient access method for similarity search in metric spaces. In VLDB, 1997.

[10] G. Costa, M. Marin, and N. Reyes. Parallel query processing on distributed clustering indexes. Journal of Discrete Algorithms, 7:3-17, 2009 (Elsevier).

[11] D. Hendler and N. Shavit. Non-blocking steal-half work queues. In PODC, pages 280-289. 2002.

[12] F. Falchi, C. Lucchese, S. Orlando, R. Perego and F. Rabitti. Caching content-based queries for robust and efficient image retrieval. In Proc. of EDBT, pages 780-790. Saint-Petersburg, Russia. 2009.

[13] G. Hjaltason and H. Samet. Index-driven similarity search in metric spaces. ACM Trans. on Database Systems, 28(4):517-580, 2003.

[14] R. Hoffmann and M. Korch and T. Rauber. Performance Evaluation of Task Pools Based on Hardware Synchronization. In Supercomputing Conference. 2004.

[15] M. Marin, R. Uribe, and R. Barrientos. Searching and updating metric space databases using the parallel EGnat. In International Conference on Computational Science (1), pages 229-236, 2007.

[16] M. Marin, R. Paredes and C. Bonacic, "High-Performance Priority Queues for Parallel Crawlers", In 10th ACM International Workshop on Web Information and Data Management (WIDM 2008), California, US, Oct. 30, 2008.

[17] M. Marin, V. Gil-Costa, and R. Uribe. Hybrid Index for Metric Space Databases. In International Conference on Computational Science, pages 327–336, 2008.

[18] J. Reinders. Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism. OReilly (2007).

[19] H. Samet. Foundations of Multidimensional and Metric Data Structures. Morgan Kaufmann, New York, 2006.

[20] L.G. Valiant. A bridging model for parallel computation. Comm. ACM, 33:103–111, Aug. 1990.

[21] P. Zezula, G. Amato, V. Dohnal, and M. Batko. Similarity Search - The Metric Space Approach. Advances in Database Systems, vol. 32. Springer, 2006.