Hybrid Architecture for Metric Space Searches

Fabricio H. Bustos[1], Marcelo Alaniz[1], Veronica Gil-Costa[1 2] and A. Marcela Printista[1 2]

[1] Laboratorio de Investigación y Desarrollo en Inteligencia Computacional
Universidad Nacional de San Luis
[2] CONICET CCT-San Luis-Argentina
{mprinti}@unsl.edu.ar

Abstract. Every day, new technologies are developed to combine the facilities arranged for shared memory systems with the facilities that provide distributed memory systems. This paper proposes a hybrid system that enables communication between threads running in a shared memory environment and a cluster of computers. To do this we use specific directives provided by MPI to solve a problem of similarity search on metric spaces .This work is part of a larger project that deals with improving query searches over high dimensional spaces, managing large volumes of data, reducing the number of distance evaluations and query response times. While the proposal of this work may be generalized and used for other problems, the results show that the proposed hybrid algorithm allows a significant improvement.
This work is part of a larger project that deals with improving the execution of parallel algorithms using a hybrid architecture. The goal is to take advantage of the features and facilities provided by the new parallel architectures that combine distributed and shared memory systems. The former allows to solve large scale problems while the second allows better use of resources.

Keywords: Hybrid algorithm, metric space, search for similar objects.

1 Introduction

New applications for search engines demand the use of data more complex than plain-text. *Metric spaces* have been proven to be useful and practical for performing similarity search on very-large collections of complex data objects such as images or audio. In this case, queries are represented by objects of the same type to those stored in the database where, for example, one is interested in retrieving the top-k objects which are the most similar to a given query. The degree of similarity between two objects is calculated by an application-dependent function called the *distance function*, which is usually expensive to compute, and pre-computed distances are used to index the database in order to reduce the average number of calls to this function during search.

Typically a large search engine is composed of a broker machine and a collection of P search nodes or processors forming a distributed memory cluster system. The broker is in charge of sending queries to processors for results calculation. Each processor is seen as a search node which is in charge of a fraction of the whole text collection. Efficient search is supported by an index data structure that is distributed onto the P processors and parallel search query processing is performed by sending the query to different processors where in each processor the arriving job can be processed by using several CPUs. For systems under heavy query traffic it is critical to reduce the number of processors hit by queries and yet to maintain an efficient throughput (defined as

number of queries entirely solved per unit of time). Smart index distribution onto processors and cache memories can be used to achieve this goal [9, 5].

Recently, the hybrid model has begun to attract more attention for at least two reasons. The first is that it is relatively easy to pick a language/library instantiation of the hybrid model; in this work we used OpenMP+MPI, solids commercial products with implementation from multiples vendors. The second reason is that several scalable parallel computers now appears to encourage this model. The idea of the hybrid parallel paradigm is to exploit parallelism beyond a single level using the *threads* paradigm to exploit the multiples cores per node (with one multithreaded process per node) while using *message passing* to communicate among the nodes.

This paper studies and evaluates a multimedia search index named Sparse Spatial Selection (SSS) [1] on a hybrid architecture that combines shared memory (multicore technology) with distributed memory systems (clusters of processors). Each processor has a number of active queries Q at any instant of time. Each query is processed locally in each distributed processor and then the results are collected by the Broker machine. Therefore, the problem addressed in this work requires communication and collaboration of different processors in a cluster of computers that can handle large volumes of information. In addition, the query processing within each processor is organized to improve the cache management and RAM memory accesses. In this way, we can also improve response query times achieved on each processor of the cluster.

2 Related work

In this section, we review the properties of metric-space, the data index selected for this work and existing approaches on parallel query processing for metric-space similarity search.

2.1 Metric spaces

A *metric space* $(\mathcal{U}, d)$ is composed of a universe of valid objects $\mathcal{U}$ and a *distance function* $d : \mathcal{U} \times \mathcal{U} \rightarrow R^+$ defined among them. The distance function determines the similarity between two given objects. The goal is, given a set of objects and a query, to retrieve all objects close enough to the query. 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 $(d(x, z) \leq d(x, y) + d(y, z))$. In this setting a *database* (instance) $\mathcal{X}$ is simply a finite collection of objects from $\mathcal{U}$. The finite subset $\mathcal{X} \subset \mathcal{U}$, with size $n = |\mathcal{X}|$, is called database and represents the collection of objects.

There are two main queries of interest for this paper:

- *range search:* that retrieves all the objects $u \in \mathcal{X}$ within a radius r of the query q, that is: $R_{\mathcal{X}}(q, r) = \{u \in \mathcal{X} : d(q, u) \leq r\}$, and
- *k-nearest neighbors search:* retrieves the set $kNN_{\mathcal{X}}(q, k) \subseteq \mathcal{X}$ such that $|kNN_{\mathcal{X}}(q, k)| = k$ and, for every $u \in kNN_{\mathcal{X}}(q, k)$ and every $v \in \mathcal{X} \setminus kNN(q, k)$, it holds that $d(q, u) \leq d(q, v)$.

In this work we focus on range queries because k-NN queries can be efficiently solved using range queries [3]. 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. Some data structures for metric spaces can be classified as pivot based techniques. Pivoting techniques select some objects as pivots, calculate the distance among all objects and the pivots, and use them in the triangle inequality to discard objects during search. Many algorithms are based on this idea [2, 11, 12].

2.2 Sparse Spatial Selection (SSS)

During construction, this pivot-based strategy selects some particular objects as pivots from the collection and then computes the distance between the pivots and the objects of the database [1]. The result is a table of distances where columns are the pivots and rows the objects. Each cell in the table contains the distance between the object and the respective pivot. These distances are used to solve queries as follows. For a range query $R_{\mathcal{X}}(q, r)$ the distances between the query and all pivots are computed. The objects x from the collection that do not hold the condition $|d(p_i, x) - d(p_i, q)| \leq r$ for *all* pivots p_i can be immediately discarded due to the triangle inequality. The objects that pass this test are considered as potential members of the final set of objects that form part of the solution for the query and therefore they are directly compared against the query by applying the condition $d(x, q) \leq r$. The gain in performance comes from the fact that it is much cheaper to effect the calculations for discarding objects using the table than computing the distance between the candidate objects and the query.

A key issue for efficiency is the method employed to calculate the pivots, which must be effective enough to drastically reduce total number of distance computations between the objects and the query. To select the pivots set, let $(\mathcal{U}, d)$ be a metric space, let $\mathcal{X} \subseteq \mathcal{U}$ be a database, and let M be the maximum distance between any pair of objects in $\mathcal{X}$. The set of pivots contains initially only the first object of the collection. Then, for each objects $x_i \in \mathcal{X}$, x_i is chosen as a new pivot if its distance to every pivot in the current set of pivots is equal or greater than $\alpha \cdot M$, being α a constant parameter. Therefore, an object in the collection becomes a new pivot if it is located at more than a fraction of the maximum distance with respect to all the current pivots.

2.3 Parallel Query Search in Metric Spaces

Distributed metric-space query processing was first studied in [10]. In this seminal work, the authors presented analytical and experimental results showing that it is possible to achieve scalable performance in this application domain. The work in [10] was extended in [7] for the context of a clustering based index data structure for metric-space databases. Several alternatives for index distribution onto processors were studied, concluding that global indexing achieves better performance than local indexing. Global indexing refers to a single index that is constructed by considering the whole set of database objects and then evenly distributing the index onto the processors. In a way, global indexing helps the broker to quickly select the f processors most likely to contain the global top-k results, and makes very unlikely to go further on the remaining $P - f$ processors. However, a drawback of global indexing is the potential for processor imbalance which arises when many queries tend to hit the same few processors. Usually, user queries tend to be skewed to particular fragments of data that change in a dynamic and an unpredictable manner. The work in [9] proposes a solution to the imbalance problem of global indexing which is based on an off-line method to distribute the index onto processors. However, this requires very expensive $O(n^2)$ pre-processing of the n-sized sequential index to determine to which processors the different sections of the index are assigned. In addition, this solution fails when user queries dynamically change focus along time.

In regards with shared memory systems, the work in [4] presents and analyze different search strategies to improve running time and load balance in a multi-core architecture. The authors show that a sync/async algorithm is suitable for different query traffic.

3 Hybrid Algorithm

3.1 Data Partition

There are different techniques to partition the data collection on P processors [7, 8, 6]. The simplest technique is called local partition. This technique distributes the objects between processors and then each processor builds its own local index using local information. Another well-known technique is the global partition, which constructs a single index and then distributes it somehow among processors. This paper does not intend to study different partitioning techniques and their advantages and/or disadvantages, but we focus on combining a distributed and shared memory system. Therefore we use the first technique mentioned above, i.e. local data partition technique. But the algorithm proposed in this paper can be adapted to other types of partitions with minimal modifications.

3.2 Query processing

To process queries we have two kind of processors or nodes: a) **Broker Node:** receives queries from users and send them to the search nodes. Then it receives the results from the search nodes to obtain the final answer. b) **Search Nodes:** search for similar objects to the queries. Each node has its own local index and a data collection partition. They access the index in order to find similar objects to the queries and retrieve those similar objects to the broker node.

The broker sends a batch of queries to all the search nodes. Each node receives the queries and process them using the local *SSS* index. When each node retrieves the most similar objects to the query and sort them by distance, it sends them back to the broker. Finally, the broker collects all responses and merge them into a single answer.

3.3 Search Algorithm

This algorithm uses the MPICH library for message passing communication between participating nodes and OpenMP library to perform the query processing using a shared memory system. The algorithm parameters are: 1) *queries*: file containing the queries. 2) *queryCount*: number of queries to be processed. 3) *batchSize*: number of queries to be sent from the broker to the search node. 4) *searchThreads*: number of threads per search node. 5) *ratio*: query search radius.

Algorithm 1 shows the main MPI directives used for the search process. First we read the parameters and we initialize the MPI execution environment. We start all participating nodes in parallel based on the parameters of MPI as well as the parameters of the program. In lines 2 and 3 we define the communicators that will be used by the MPI processes. Here we use two communicators, one for sending and one for receiving messages. Then, from line 4 to 15 we declare a new type of data used for sending messages. In line 7, we declare an array of size $batchSize$ where each cell will contain information relevant to a query. In line 8 we declare a name for the data type. Then in line 9 we declare an array that indicates which are the basic types of data that makes up the structure used and the order of location within the structure. In line 10 we declare an array of integers which specifies the maximum size of each of the basic types, in this case, we use an integer and an array of 128 characters. In line 11 we declare an array of memory addresses that is initialized in lines 12 and 13. In line 14 we create the MPI data type taking the characteristics indicated in the previous lines. Basically it indicates the creation of a new data type that will contain blocks where the length of each block as well as the displacement between them and the type of each line was defined in the previous lines of code. The new type is named sendMsgType (line 8). In line 13, MPI indicates that this new type of data can be

```
1. Search_Hybrid (queries, queryCount, queryCount, searchThreads)
2.    MPI_Comm dup_comm_a //Communicator of the thread that sends messages
3.    MPI_Comm dup_comm_b //Communicator of the thread that receives messages
4. //---------Data structure used to send messages
5.    struct sendMsgStruct sendMsg[batchSize]
6.    MPI_Datatype sendMsgType
7.    MPI_Datatype typeSend[2] = {MPI_INT, MPI_CHAR}
8.    int blockLenSend[2] = {1, 128}
9.    MPI_Aint dispSend[2]
10.   dispSend[0] = 0
11.   dispSend[1] = (MPI_Aint)&(sendMsg[0].query) - (MPI_Aint)&sendMsg[0]
12.   MPI_Type_create_struct(2, blockLenSend, dispSend, typeSend, &sendMsgType)
13.   MPI_Type_commit(&sendMsgType)
14. //-------Data structure used to receive messages
15.   MPI_Datatype recvMsgType
16.   struct recvMsgStruct recvMsg[batchSize]
17.   MPI_Datatype typeRecv[3] = {MPI_INT, MPI_INT, MPI_INT}
18.   int blockLenRecv[3] = {1, 1, RESPONZE_SIZE}
19.   MPI_Aint dispRecv[3]
20.   dispSend[0] = 0
21.   dispSend[1] = (MPI_Aint)&(sendMsg[0].query) - (MPI_Aint)&sendMsg[0]
22.   MPI_Type_create_struct(2, blockLenSend, dispSend, typeSend, &sendMsgType)
23.   MPI_Type_commit(&sendMsgType)
24.   MPI_Init_thread(.. MPI_THREAD_MULTIPLE .. )
25.   MPI_Comm_size(MPI_COMM_WORLD, &nodeCount)
26.   MPI_Comm_rank(MPI_COMM_WORLD, &myRank)
27.   MPI_Comm_dup(MPI_COMM_WORLD, &dup_comm_a)
28.   MPI_Comm_dup(MPI_COMM_WORLD, &dup_comm_b)
29.   if( myRank == BROKER_PROCESS )
30.         Run_broker_code()
31.   else
32.         Run_node_code()
```

Fig. 1. Search algorithm.

used within communications messages. From line 14 to 24, we declare the data structure used to receive messages. Here we proceed in the same way as for the data type defined in the previous lines to send messages. In line 15 we define the name of data type. We declare an array that contains details of relevant responses (line 16). In line 17 we declare an array that defines the basic types of this structure, in this case consisting of 3 integers. Then in line 18 we declare an array indicating the size of each block, the two first block corresponds to a single integer, while the third block corresponds to an integer array of size $RESPONSE\_SIZE$. In lines 19, 20 and 21 we define the offset between each block. In line 22 we create the data structure. From line 23 on, MPI can use this data structure.

Line 24 shows the use of the MPI_Init_thread() function that initializes the thread environment with multiple levels to support threads (MPI_THREADS_MULTIPLE) without restriction to make calls to MPI directives. Then, one or more threads per processor can execute MPI directives concurrently. Later in line 25 the variable $nodeCount$ determines the number of running processes. In line 26 we recover the process identifier ($myrank$). In lines 27 and 28 we create

two communicators to be used. The global communicator MPI is duplicated. Finally, depending on the process identifier we run the code in line 30, which corresponds to the Broker machine, or we go to the line 32 which corresponds to the search nodes.

3.4 Design of the threads

In MPI, the message passing between processes is done using a communicator. Because we always need a communicator, MPI provides the MPI_COMM_WORLD and is the main communicator for all processes involved. To avoid conflicts when using the same communication channel, we proceeded to double the MPI communicator MPI_COMM_WORLD through the variables dup_comm_a and dup_comm_a. These variables use the same set of processes but operate in different context, allowing, for example, to send a batch of queries and to receive an answer from one of the search nodes at the same time. Below are the specific tasks of each node.

The Broker node uses OpenMP directives to create two threads, one for sending queries and another for receiving query results from the search nodes as shown in Algorithm 2. To do so, each thread gets its identifier in line 3. Since this process is running on a multicore processor, we assign each thread to a different core allowing a higher degree of parallel execution. The allocation of threads into cores is done using the affinity mask $sched\_setaffinity()$ as shown in line 7. Thus, it makes better use of multicore processor and two threads perform their tasks on separate cores largely avoiding the overhead that is created when making an exchange of context in the case of two threads running on a single core.
The following describes the Broker threads called Thread B_0 and Thread B_1.

- **Thread B_0**: Reads the query, then iterates over for delivering a message with a query batch determined by the parameter $batchSize$. That message is sent to the search nodes by a blocking broadcast in each iteration. To this end, lines 8 through 16 of Algorithm 2 are executed by the Thread B_0 responsible for sending queries. The iterations are made between lines 9 and 15. Then we use the MPI_Bcast in line 14 to send queries via a broadcast operation. Note that the communicator uses dup_comm_a defined in Algorithm 1.
- **Thread B_1**: Iterates until it receives all the answers from all search nodes. The code in this thread is between lines 17 to 25. In each iteration, the thread keeps waiting for a reply message using the MPI_Probe directive (line 21). Upon a reception of a message the thread identifies the search node that sent it and processed the responses recording statistics. The reception of the message is done through the MPI_Recv directive in line 22.

The search nodes use MPI to assign a process identification number (tid), which is also used to identify the partition and the index used to process queries. Threads are created with the OpenMP directive $parallel$ indicating the number of threads to be create, as well as the shared data between them. We classify the threads into two groups: a) threads of communication and b) search threads. The threads of communication work as the broker threads, one to send messages and other to receive messages using different communicator variables. They communicate with the Broker node for receiving queries and send the results. The number of search threads is indicated by the parameter $searchThreads$.

Since the search nodes are also multicore processors, threads are created and allocated into cores using the $sched\_setaffinity()$ function, but taking into account the following conditions:

```
1.  Run_broker_code()
2.     #pragma omp parallel num_threads(2)
3.     tid ← omp_get_thread_num()
4.     cpu_set_t mask
5.     CPU_ZERO (&mask)
6.     CPU_SET (tid, &mask)
7.     sched_setaffinity(.. &mask ..)
8.     if ( tid==0 ){
9.             while( queryCount > 0 ) {
10.                    while( i< batchSize ){
11.                           q← get_query
12.                           Msg ← add_query
13.                    }
14.                    MPI_Bcast(Msg, batchSize, sendMsgType, BROKER_PROCESS, dup_comm_a)
15.            }
16.    }
17.    else {
18.            MPI_Status status
19.            total ← number expected of messages
20.            while ( total > 0) {
21.                    MPI_Probe(MPI_ANY_SOURCE ... dup_comm_b, &status)
22.                    MPI_Recv(.. dup_comm_b, &status ..)
23.                    Process the message received and to obtain statistics
24.            }
25.  }
```

Fig. 2. Algorithm executed by the broker machine.

the threads of communication are allocated into the first two cores and the other threads are distributed among the remaining cores.

The data shared by the threads is listed in the clause $shared$ of the OpenMP $parallel$ directive. These data remain in shared memory and can be accessed by concurrent threads. The data are:

- A FIFO input queue, where incoming queries are located and not yet processed.
- An output FIFO where to place the results of queries that must then be sent to the Broker node.
- They also share the partition of the collection of objects and the local index.

While these data are shared by the threads, the queue access is restricted because they have the condition that only one thread can remove or insert an object in the queue at a time. This restriction is achieved indicating critical regions of code with the OpenMP directive $critical$. Thus, we obtain a similar behaviour to a blocking queue.

OpenMP identifies each thread with a number that is obtained with the routine $omp\_get\_num\_thread()$. As mentioned above, identifying the threads not only allows us to allocate them into the corresponding cores but also indicates the functionality of each thread.

– **Thread S_0**: It is the thread that receives batch of queries sent from the Broker node. In each iteration the thread waits for a broadcast message containing a query batch. Upon receiving each query removes the batch and add it to the input queue so they can be taken by the search threads. It uses a duplicated communicator and dup_comm_a.

– **Thread S_1**: It is the thread responsible for sending the query answers. The thread loops through the output queue by removing the answers and put them together into a message that is sent to the Broker using the routine $MPI\_Send()$. It uses a duplicated communicator and dup_comm_b.

– **Thread S_n**: These threads process the queries. So they have access to both the input queue, to take new queries, as well as to the output queue where they leave the answers. When they remove a query from the input queue the threads proceed to search for similar objects by applying the SSS algorithm as mentioned before, using the partition data collection and the local index. Once they get the answers for the given query, those answers are packaged to be sent to the broker node.

As we saw, the queues are FIFO data structures that are used for communication between the threads of a search node and on the other hand, the message passing is used to process communication between the nodes. Fig. 3 shows the communication between the search nodes and the Broker node as well as the communication that takes place between the threads.

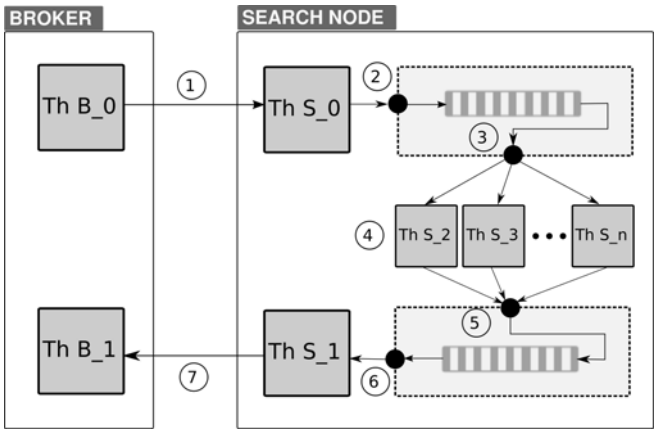

Fig. 3. System architecture: communication between process and threads. 1)The broker node sends a messages with queries. 2) Thread S_0 receives the messages, obtains the queries and put them into the input queue, 3) the search threads access to the queue to obtain a query, 4) each search thread process a different query and 5) stores the results into the output queue, 6) Thread S_1 obtains the results, and 7) those results are sent to the broker.

4 Experimental Results

4.1 Experimental Settings

This paper used three collections to evaluate the performance of the proposed search algorithm. For the collections made up of words we use the edit distance to compute the similarity between two terms. The collections are: a)*Spanish*: with 51.589 words from a Spanish dictionary. The maximum distance between two words is 21. b) *English*: Consisting of 69.069 English words where the longest word in this collection has 21 characters. c) *UK*: Consisting of 200,000 words from a sample of 1.5TB from the UK Web. Queries on all collections were selected at random from the same database.The experiments were performed in the WS-49 cluster of the UNSL which consists of 16 CPUs of 64 bits with Intel Q9550 Quad Core 2.83GHz processors and RAM memory of 4GB DDR3 1333Mz. 9 CPUs of 64 bits with Intel E6750 Core 2 Duo 2.66GHz processors and RAM memory 2GB DDR2 667Mz.

The broker node was executed on a dual core processor and the search nodes were executed on the remaining 16 Quad Core processors which are responsible for processing queries. To analyze the system we performed different executions for different numbers of processors, number of search threads, message size, etc. which are shown in Table 1 and will be referenced for subsequent figures.

	N	Th	Bs
C1	2	1	32
C2	2	1	64
C3	2	1	128
C4	2	1	256
C5	2	2	32
C6	2	2	64
C7	2	2	128
C8	2	2	256
C9	2	4	32
C10	2	4	64
C11	2	4	128
C12	2	4	256
C13	2	8	32
C14	2	8	64
C15	2	8	128
C16	2	8	256

	N	Th	Bs
C17	4	1	32
C18	4	1	64
C19	4	1	128
C20	4	1	256
C21	4	2	32
C22	4	2	64
C23	4	2	128
C24	4	2	256
C25	4	4	32
C26	4	4	64
C27	4	4	128
C28	4	4	256
C29	4	8	32
C30	4	8	64
C31	4	8	128
C32	4	8	256

	N	Th	Bs
C33	8	1	32
C34	8	1	64
C35	8	1	128
C36	8	1	256
C37	8	2	32
C38	8	2	64
C39	8	2	128
C40	8	2	256
C41	8	4	32
C42	8	4	64
C43	8	4	128
C44	8	4	256
C45	8	8	32
C46	8	8	64
C47	8	8	128
C48	8	8	256

Table 1. Configuration for different experiments. The column **N** indicates the number of search nodes, **Th** the number of search threads and **Bs** indicates the number of queries in each message.

Running Times First we compute the running time for each of the settings of Table 1 for all three data collections. The results are shown in Fig. 6 for the Spanish, English and UK collections, which is plotted separately because the running times are higher with this last collection. The three collections present a tiered effect by varying the settings. This is due mainly to the increasing e amount of search nodes and the number of threads used in each processor. The most significant jumps occur with the configurations C5, C21 and C37 and are produced because we increase the number of threads and we use a small batch size. Namely, the running time reported

with 4 processors with a single search thread was similar to the running time reported with 2 or 4 processors and 8 threads. A substantial improvement occurs when increasing from 1 to 2 threads when we have 4 processors. Something similar occurs with the configuration C37 (with 8 processors and 2 threads). In the first jump located between configurations C4 and C5, the improvement in execution times is produced by increasing the amount of thread.

Batch Size An important parameter to analyze is the size of the batch of queries that are sent in each message. In Fig. 4 we can see the behavior produced by the batch size using the UK collection. In Fig. 5**(a)** we can see the execution time using a batch size of 32 queries. In Fig. 5**(b)** we can see in contrast, the behavior using a batch size of 256 queries. In both cases there is a significant improvement by increasing the size of threads from one to two, then for a greater number of threads the system remains stable.

Number of Threads and Nodes The next parameter to consider is the number of search threads used per node. As we saw in Fig.6 a significant improvement is obtained when switching from one thread to two threads per node. Fig.7**(a)** shows the execution times using different numbers of threads for a batch size of 32 queries and Fig.7**(b)** using a batch size of 256 queries. Both figures were obtained from the execution of the algorithm with the UK collection. Using more than one thread has a significant impact in a system performance. Note also that there is almost no difference when using 2 or 4 threads, however an improvement is achieved by using 8 threads with two search nodes. This shows that by using more threads a major improvement in execution time is achieved. Keep in mind that we can not add an infinite number of threads because they consume time in context switching.

Finally, Fig.8 shows the running time obtained with the UK collection and processing 100000 queries. This figure shows normalized times to better illustrate the behaviour of the algorithm. To this end we divide the observed values by the maximum. Increasing the number of threads has the same effect with P=4 and P=8. We can reduce the running time by 5%. With more threads the execution time is slightly decreased, but the results do not show a significant improvement. On the other hand, doubling the number of nodes or processors we reduce running time by 80%.

5 Conclusions and Future Work

In this work we developed an algorithm that combines the facilities provided by distributed systems and shared memory systems. To this end, we have combined the use of two parallel programming libraries: MPI and OpenMP. The results show an important improvement when using more than one thread per node, but with two threads or more the improvement is minimal. As future work will investigate load balancing and scheduling algorithms to allow better use of resources and avoid competition.

Acknowledgments

We wish to thank to the Universidad Nacional de San Luis, the ANPCYT and the CONICET from which we receive continuous support.

References

1. N. Brisaboa, A. Faria, O. Pedreira, and N. Reyes. Spatial selection of sparse pivots for similarity search in metric spaces. *Journal of Computer Science & Technology*, 7(1), 2007.

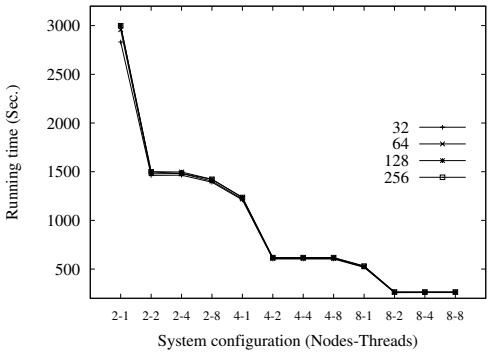

Fig. 4. Running time for different batch sizes.

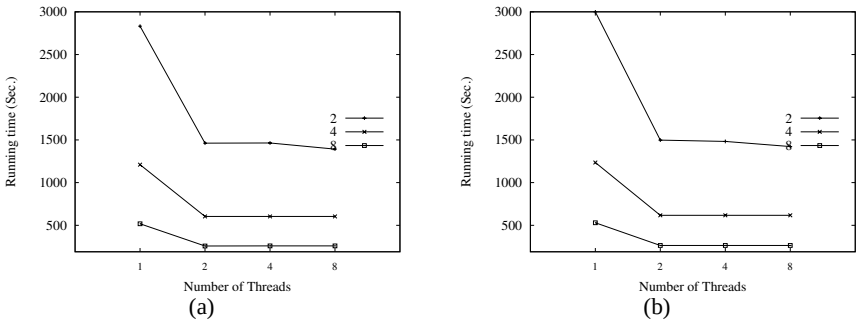

Fig. 5. Running time with different number of threads *a)* batch size = 32; *b)* and batch size = 256.

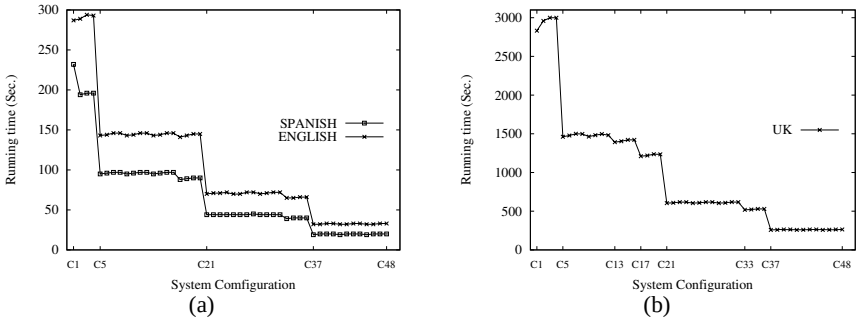

Fig. 6. Running time reported with the*a)* Spanish and English collections; *b)* UK collection.

2. E. Chavez, J. Marroquin, and G. Navarro. Fixed queries array: A fast and economical data structure for proximity searching. *MTA*, 14(2):113–135, 2001.

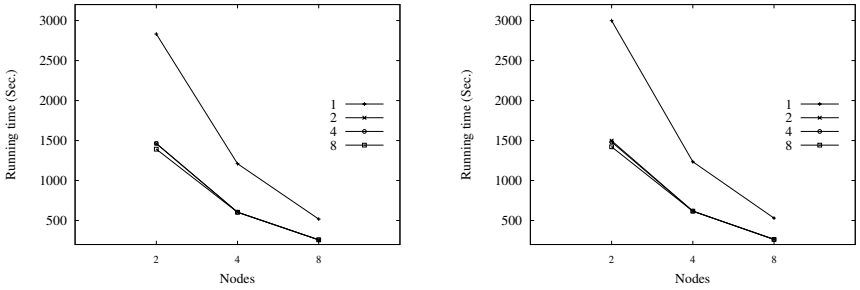

Fig. 7. Running time for different number of nodes *a)* with a batch size =32; *b)* and a batch size = 256.

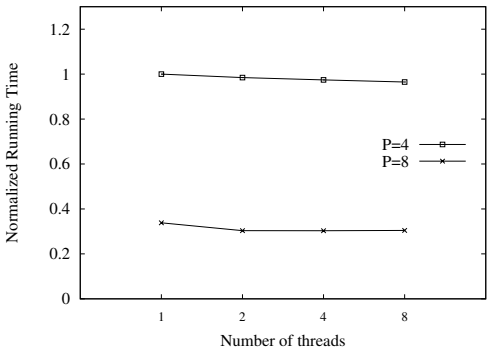

Fig. 8. Normalized running time for P=4 and P=8 nodes with different number of threads

3. E. Chavez and G. Navarro. A compact space decomposition for effective metric indexing. *PRL*, 26(9):1363–1376, 2005.
4. Veronica Gil Costa, Ricardo J. Barrientos, Mauricio Marín, and Carolina Bonacic. Scheduling metric-space queries processing on multi-core processors. In *PDP*, pages 187–194, 2010.
5. Fabrizio Falchi, Claudio Lucchese, Salvatore Orlando, Raffaele Perego, and Fausto Rabitti. Caching content-based queries for robust and efficient image retrieval. In *EDBT*, pages 780–790, 2009.
6. V. Gil-Costa and M. Marin. Distributed sparse spatial selection indexes. *In 16th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP 2008),IEEE-CS Press*, pages 440–444, Toulouse, France, Feb. 13-15, 2008.
7. V. Gil-Costa, M. Marin, and N. Reyes. Parallel query processing on distributed clustering indexes. *Journal of Discrete Algorithms (Elsevier)*, 7:3–17, 2009.
8. M. Marin, V. Gil Costa, and R. Uribe. Hybrid index for metric space databases. In *ICCS*, pages 327–336, 2008.
9. M. Marin, F. Ferrarotti, and V. Gil-Costa. Distributing a metric-space search index onto processors. In *ICPP*, pages 13–16, 2010.
10. N. Papadopoulos and Y. Manolopoulos. Distributed processing of similarity queries. *Distrib. Parallel Databases*, 9(1):67–92, 2001.
11. E. Vidal. An algorithm for finding nearest neighbors in (approximately) constant average time. *Pattern Recognition Letters*, 4:145–157, 1986.
12. P. Yianilos. Locally lifting the curse of dimensionality for nearest neighbor search. In *11th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 361–370, 2000.