New technologies for big multimedia data treatment

Mercedes Barrionuevo, Luis Britos, Fabricio Bustos, Verónica Gil-Costa,
Mariela Lopresti, Virginia Mancini, Natalia Miranda, Cesar Ochoa,
Fabiana Piccoli, A. Marcela Printista, Nora Reyes
Laboratorio de Investigación y Desarrollo en Inteligencia Computacional (LIDIC).
Departamento de Informática.
Universidad Nacional de San Luis.
Argentina.
e-mail: {*gvcosta, mpiccoli, mprinti, nreyes*}*@unsl.edu.ar*

***Abstract*—With the technology advance and the growth of Internet, the information that can be found in this net, as well as the number of users that access to look for specific data is bigger. Therefore, it is desirable to have a search system that allows to retrieve information at a reasonable time and in an efficient way. In this paper we show two computing paradigms appropriate to apply in the treatment of large amounts of data consisting of objects such as images, text, sound and video, using hybrid computing over MPI+OpenMP and GPGPU. The proposal is developed through experience gained in the construction of various indexes and the subsequent search, through them, of multimedia objects.**

***Index Terms*—Metric Space, Hybrid Computation, GPU, Index, Parallel Searching**

I. Introduction

Metric spaces have been proven to be a useful and practical model for similarity search problems 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.

Most existing search structures have been designed to run on a single computer. They are built with different assumptions about type of distance function, form of query, index storage and temporal properties of the data to be organized. These centralized metric indexes achieve a significant speedup when compared to the sequential scan, but their costs increases linearly with the growth of the dataset [1]. Thus, the ability of centralized indexes to maintain a reasonable query response time when the datset multiples in size, its scalabilty, is limited.

To tackle this problem we propose to use parallel and distributed algorithms that aim to optimize resource utilization, response time and throughput. The field of architecture and paradigms for parallel and distributed computation environment is large due to the numerous research challenge it offers for different objectives. Recently, the hybrid architecture 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 and GPGPU, 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.

The remaining of this paper is organized as follows: section II describes the concepts required for the development of this work. Sections III and IV develop, respectively, hybrid programming and GPU approaches applied to the metric spaces context. Section V locates the concepts of cloud and HPC applied to the processing of big multimedia data treatment. Sections VI discusses some final considerations.

II. Previous Concepts

In this section, we explain the main concepts to develop this work.

A. Metric Spaces, Queries and Indexes

A metric space (X, d) is composed of a universe of valid objects X and a distance function $d : X \times X \rightarrow R^+$ defined among them. The distance function determines the similarity (or dissimilarity) between two given objects and

satisfies several properties such as strict positiveness (except $d(x,x) = 0$, which must always hold), symmetry ($d(x,y) = d(y,x)$), and the triangle inequality ($d(x,z) \leq d(x,y) + d(y,z)$). The finite subset $U \subseteq X$ with size $n = |U|$, is called the *database* and represents the set of objects of the search space. The distance is assumed to be expensive to compute, hence it is customary to define the search complexity as the number of distance evaluations performed, disregarding other components.

There are two main queries of interest [2], [1], [3]: Range Searching and the k Nearest Neighbors (k-NN). The goal of a range search $(q,r)_d$ is to retrieve all the objects $x \in U$ within the radius r of the query q (i.e. $(q,r)_d = \{x \in U / d(q,x) \leq r\}$). In k-NN queries, the objective is to retrieve the set k-NN$(q) \subseteq U$ such that $|k - NN(q)| = k$ and $\forall x \in k-NN(q), v \in U \wedge v \notin k-NN(q), d(q,x) \leq d(q,v)$.

When an index is defined, it helps to retrieve the objects from U that are relevant to the query by making much less than n distance evaluations during searches. The saved information in the index can vary, some indices store a subset of distances between objects, others maintain just a range of distance values. In general, there is a tradeoff between the quantity of information maintained in the index and the query cost it achieves. As more information an index stores (more memory it uses), lower query cost it obtains. However, there are some indices that use memory better than others. Therefore in a database of n objects, the most information an index could store is the $n(n-1)/2$ distances among all element pairs from the database. This is usually avoided because $O(n^2)$ space is unacceptable for realistic applications [4].

Proximity searching in metric spaces usually are solved in two stages: preprocessing and query time. During the preprocessing stage an index is built and it is used during query time to avoid some distance computations. Basically the state of the art in this area can be divided in two families [2]: *pivot based algorithms* and *compact partition based algorithms*. In the first case, the index consists in a set of pivots $\{p_1, \ldots, p_m\} \subseteq U$, which computes and keeps (in a data structure, usually like a tree) some (or all) distances $\{d(p_1,x), d(p_2,x), \ldots, d(p_m,x)\}, x \in U$. The queries are solved considering all pivots. In the second case, the space is divided into small and compact zones. A set of objects, called *centers*, $\{c_1, \ldots, c_s\} \subseteq U$ are chosen and the rest of the elements are distributed into the s zones defined in different ways by the centers c_i. The index is composed by the centers, the elements of each zone, and often some additional distances.

There is an alternative to "exact" similarity searching called approximate similarity searching [5], where accuracy or determinism is traded for faster searches [2][1][3][6], and encompasses *approximate* and *probabilistic algorithms*. The goal of approximate similarity search is to reduce *significantly* search times by allowing some errors in the query outcome.

In approximate algorithms one usually has a threshold ϵ as parameter, so that the retrieved elements are guaranteed to have a distance to the query q at most $(1+\epsilon)$ times of what was asked for [7]. Probabilistic algorithms on the other hand state that the answer is correct with high probability. Some examples are [8], [9]. In the next section we detail a probabilistic method: *Permutation Index* [10].

B. Big Data

Many institutions and organizations today produce a large amount of data per day [11]. Recent statistics show that Facebook [1] has more than 900M users, 2.5 billion content items and produces more than 105 TB per hour. Twitter [2] is another source of large amount of data producing more than 800 tweets per second. Every minute more than 60 blogs are created, 168 million mails are sent, 600 new videos and so on. This situation has given rise to the existence of the Big Data problem.

Big data is important due to more data may lead to more accurate analyses. More accurate analyses may lead to more confident decision making. And better decisions can mean greater operational efficiencies, cost reductions and reduced risk. However, some problems arise with this new exponential growth and availability of data. (1) The data cannot be easily analyzed in on a simple laptop (say few Gigabytes to Terabytes). (2) Most organizations do not know whether it is worth keeping the data or not. (3) How to store/organize the new data. (4) Visualization is another important issue as most scientific and commercial applications require to plot the data. (5) How to find out which data points are really important. (6) The data are generally quirky and messy (unstructured text, json[3] files with lots of missing data, fast files with quality metrics, etc.)

This large amount of data consist of text and also multimedia data like sound and video. In fact, most of the current data is not in a structured format. For example, blogs and tweets are weakly structured texts, while images and video are only structured for storage and display, but totally unstructured with respect to their semantic content. As it is the content which is important for most applications (specially web applications), its extraction in a structured way is a major challenge.

C. Hybrid Computation

Recent computational architectures, such as multi-core CPUs and clusters, proposed several orders of magnitude faster than the corresponding serial implementations. To achieve these speedups they have imposed additional complexities on programming. Programmers must take the initiative to implement parallel processing capabilities to their programs to fully utilize the hardware available.

[1] www.facebook.com
[2] www.twitter.com
[3] JavaScript Object Notation

The current infrastructures have distributed-memory but use shared-memory. As a result of efforts of MPI designer to deliver efficient MPI implementations, which put the full capabilities of shared-memory system to use for high-performance intra-node message passing, most parallel applications still use *pure MPI* for parallelization. This means that in the era of multi-core, many applications run an MPI process per core. From the point of view of the programmer, the model known as *pure MPI* ignores the fact that the cores within a processor shares memory. In this model, it is not required that the MPI library supports multi-thread, which simplifies the algorithm implementation. All communications among processes inside of same processor are performed by producing an MPI message exchange in the application. This communication has to be optimized by using methods supporting shared memory among the process running on the same processor and through the interconnection network of cluster for MPI processes running on different processors.

The work presented in [12] questions whether nowadays it is appropriate to continue using these *pure MPI* trend. A new style considers a hybrid OpenMP/MPI programming model which allows any MPI process to spawn a team of OpenMP threads. Thus, inserting OpenMP compiler directives into an MPI code is a straightforward way to build a hybrid parallel program.

The idea of hybrid parallel programming is to decompose an application into tasks and the use of multiple control flows running on different processors or cores to reduce the runtime. An important feature is that threads within a process share the address space, i.e. they have a common address space. In particular, different threads of a single process can be assigned to different cores of a multi-core processor.

Based on the above, a suitable choice for a parallel programming model becomes extremely important on current hybrid architectures. The taxonomy of parallel programming models proposed in [13] defines the following levels: a) *Pure MPI* where each core is used for one MPI process; b) *Pure OpenMP* based on virtual distributed shared memory system (DSM) where the application is parallelized only with shared memory directives; c) *Hybrid MPI+OpenMP Master-only* where there is no overlap among message-passing MPI calls and application code in other threads; and d) *Hybrid MPI+OpenMP with Overlap* where one or more threads of the OpenMP team to execute communication, letting the rest do the actual computations.

Moving from a single thread scheme to a multi-thread scheme is not a simple task from the perspective of communication library. To support an implementation of a multi-thread scheme, the MPI-2.2 standard defines four levels to initialize MPI thread environment. The argument required by the MPI_INIT function is used to specify the desired level of thread. The possible values are listed in increasing order of thread support:

1) MPI_THREAD_SINGLE: only one thread will execute;
2) MPI_THREAD_FUNNELED: the process may be multi-threaded, but only the main thread will make MPI calls (all MPI calls are “funneled” to the main thread);
3) MPI_THREAD_SERIALIZED: the process may be multi-threaded, and multiple threads may make MPI calls, but only one at a time: MPI calls are not made concurrently from two distinct threads (all MPI calls are “serialized”) and
4) MPI_THREAD_MULTIPLE: multiple threads may call MPI, with no restrictions.

D. GPGPU

Mapping general-purpose computation onto GPU implies to use the graphics hardware to solve any applications, not necessarily of graphic nature. This is called GPGPU (General-Purpose GPU), GPU computational power is used to solve general-purpose problems [14], [15]. The parallel programming over GPUs has many differences from parallel programming in typical parallel computer, the most relevant are: *The number of processing units*, *CPU-GPU memory structure* and *Number of parallel threads*.

Every GPGPU program has many basic steps, first the input data transfers to the graphics card. Once the data are in place on the card, many threads can be started (with little overhead). Each thread works over its data and, at the end of the computation, the results should be copied back to the host main memory.

Not all kind of problem can be solved in the GPU architecture, the most suitable problems are those that can be implemented with stream processing and using limited memory, i.e. applications with abundant parallelism.

The Compute Unified Device Architecture (CUDA), supported from the NVIDIA Geforce 8 Series, enables to use GPU as a highly parallel computer for non-graphics applications [14], [16]. CUDA provides an essential high-level development environment with standard C/C++ language. It defines the GPU architecture as a programmable graphic unit which acts as a coprocessor for CPU. It has multiple streaming multiprocessors (SMs), each of them contains several (eight, thirty-two or forty-eight, depending GPU architecture) scalar processors (SPs).

The CUDA programming model has two main characteristics: the parallel work through concurrent threads and the memory hierarchy. The user supplies a single source program encompassing both host (CPU) and *kernel* (GPU) code. Each CUDA program consists of multiple phases that are executed on either CPU or GPU. All phases that exhibit little or no data parallelism are implemented in CPU. Contrary, if the phases present much data parallelism, they are coded as *kernel* functions in GPU. A *kernel* function defines the code to be executed by each thread launched in a parallel phase.

GPU computation considers a hierarchy of abstraction layers: $grid$, $blocks$ and $threads$. The $threads$, basic execution unit that executes *kernel* function, in the CUDA model are grouped into $blocks$. All threads in a block execute on one SM and communicate among them through the *shared memory*. Threads in different blocks can communicate through *global memory*. Besides shared and global memory, the threads have their local variables. All $Thread - blocks$ form a $grid$. The number of grids, blocks per grid and threads per block are parameters fixed by the programmer, and adjustable to improve performance.

Respect of memory hierarchy, CUDA threads may access data from multiple memory spaces during their execution. Each thread has private local memory and each block has shared memory visible to all its threads. These memories have the same lifetime that the *kernel*. All threads have access to the same global memory and two additional read-only memory spaces: the constant and texture memory spaces, which are optimized for different memory usages. The global, constant and texture memory spaces are persistent across launched *kernel* by the same application. Each kind of memory has its own access cost, and the global memory accesses are the most expensive.

III. Metric Spaces using Hybrid Computation

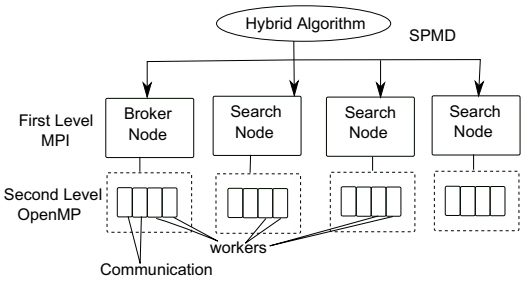

Fig. 1. Two level hybrid algorithm approach. In the first level communication is managed with the MPI library. In the second level communication is managed with the OpenMP library.

Figure 1 shows a hybrid approach based on the Single Program Multiple Data (SPMD) programming model. The hybrid approach has two programming levels. The first level is used to schedule distributed computation and to perform intra-node communication by means of the MPI library. The second level, is used to manage thread computation and inter-node communication by means of the OpenMP library.

In particular, Figure 2 presents the thread management scheme proposed in [17] for a pivot-based metric space index named Sparse Spatial Selection (SSS) [18]. This approach uses the MPICH library for inter-node communication and OpenMP library to perform the query processing using shared memory. The application architecture considers $P + 1$ MPI processes, P Query Processing processes and the Broker process, distributed on $P + 1$ physical nodes. MPI was initialized to support the thread environment (MPI_THREADS_MULTIPLE) where multiples threads may call MPI, with no restrictions.

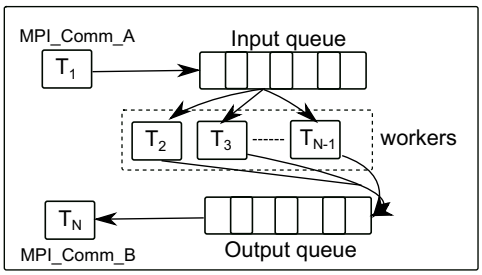

Fig. 2. Thread management in a hybrid parallel platform.

Inter-node messages are handled by two threads of a single MPI process by means of the MPI_COMM_WORLD communicator. In other words, the communication is done by the two specific threads, one to receive messages and another to send messages. Those threads use an input and output queue protected by critical regions. Queues are FIFO data structures that are used as an asynchronous method of intra-node communication. Moreover, each thread has a different MPI_Comm communicator to perform communication (send/receive messages) in parallel. All other non-communicating threads in a node are executing the index search and distance evaluations. We call them *worker* threads.

Threads are created and allocated into cores using the system call sched_setaffinity(), but taking into account the following conditions: the threads of communication are allocated into the first two cores and the other threads are distributed among the remaining cores of the node.

1) Distributed Memory Management: Building a metric space index over the parallel platform described above can be made as follows.

At a high intra-node programming level, the index can be build using a local, global or even mixed partitioned approach [19]. By using a local partitioning approach the whole database is evenly distributed among nodes and then each node builds its own index using the local data. The index construction phase has a low cost as no communication among nodes is required. But during the query processing phase resource utilization is wasted as every single query has to be sent to all search nodes and more communication and computations is required per query. Therefore at any instant time there is only one query being solved.

The global partitioned approach builds a single sequential index using the whole database collection. Then the index is distributed among nodes using different criteria. E.g. the SSS index can be partitioned by rows or by columns as shown in [17]. The index construction phase is expensive in terms of memory allocation and communication specially for large database collections. But during the query processing

phase resourses tend to be well used due to a single query visits few nodes. Therefore at any instant time more than one query can be solved. As a side effect communication and query response time are reduced.

A mixed partitioning approach typically uses global information (such as centers or pivots selected from the whole database) but each node builds its own local index using the global information. This approach tends to use the best features of both local and global index partitioning.

2) Shared Memory Management: At a low inter-node programming level, each node has a single index partition. All threads access to the same main memory so it makes no sense to partition the index again among threads. Then the question at this point is how to assign the resources (threads) to process the incoming queries. A first approach named *Bulk* [20] all threads work together to solve each query. To this end, the query processing operation is divided into small tasks containing information of the specific job assigned to a thread such as the next cell/node of the index to be examined. A task usually involves computing distance evaluations and 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. Each thread has a private local requirement queue and a secondary requirement queue that maintain tasks. This second queue is accessed by other threads to search for tasks. Then, this approach requires periodically synchronizations to avoid read-write conflicts among threads.

A second approach named *Local* [20] each query is solve by a single thread. Neither data sharing nor periodical synchronizations are required because each thread completely processes a query by using the sequential algorithm.

IV. Metric Spaces using GPU

The computational capabilities of many-core GPUs have been exploited to improve the query process of metric spaces. There are many massively parallel algorithms for metric indexes implemented in a GPU. Querying for k-NN has obtained most of the attention of researchers in the area. In [21], [22], [23], [24], [25], [26], [27] improve explicity the brute force algorithm (or sequential scan) to find the k-NN. They differ in the parts parallelized or the methodology applied. Other works [21], [28], [29] implemented some well known sequential metric indexes, such as the List of Clusters (LC) and the SSS-Index. For the case of vector data authors in [30], [31], [32] use Kd-trees for finding the k-NN and [30] apply a variant of the Kd-tree for the all k-NN problem.

All algorithms in the literature [21], [29], [22], [23], [24], [25], [26], [31], [28] for k-NN using GPU, solutions have high complexity in the data structures. Furthermore, they have a high granularity. Kernels are not uniform and have a lot of branching. This implies synchronization and serialization of the threads, which means all of them have to wait to be in the same path again to resume. In a nutshell, they use conditionals and do diverse tasks depending on comparisons. On the other hand the algorithms demand a lot of memory resources for the data structures and intermediate data, e.g. distances to pivots, and allocate only very small instances of the metric databases. For example in [29] they use only one thread block for the actual k-NN search, this implies overloading all the threads in the block and consequently suboptimal GPU resources usage, most of the threads are not used. In [23], [24], [25] they propose to solve several queries at a time, but they use just the same amount of threads than for a single query. This again implies thread overload, memory starvation and idle processing units in the GPU. In [26] is also suboptimal in resource usage, to the point of letting a single thread to finish the searching process, implying all other threads are idle.

In [33], [34], [35], we have learned from all the above examples, and also in these proposals, we have tailored solutions which are uniform and maximize the GPU usage. We have (1) carefully selected the number of threads, (2) acceded to memory in coalescent form, (3) maximized the use of shared memory, and (4) taken adventaje of zero cross talk among threads because data independence. Additionally our proposals have zero overload in the data structures, which implies all the available memory can be used for the database. We work in two different queries types: "exact" similarity search [35] and "approximate" similarity search [33], [34]. Beside, as in large-scale systems such as Web Search Engines indexing multimedia content, it is critical to deal efficiently with streams of queries rather than with single queries. Therefore, it is not enough to speed up the time to answer only one query, but it is necessary to leverage the capabilities of the GPU to parallely answer several queries, our proposals can solve many queries at the same time.

In order to answer parallely many queries, GPU receives the queries set and it has to solve all of them. Each query, in parallel, applies the process, therefore the number of needed resources for this is equal to the resources amount to compute one query multiplied the number of queries solved in parallel. The number of queries to solve in parallel is determined according to the GPU resources mainly its memory. If q are parallel queries, m the needed memory quantity per query and i the needed memory by permutation index, $q * m + i$ is the total required memory to solve q queries in parallel. Solving many queries in parallel involves carefully manage the blocks and their threads. At the same time, blocks of different queries are accessed in parallel. Hence it is important a good administration of threads: which query it is solved and which database element it is responsible. The task is possible by establishing a relationship among *Thread Id*, *Block Id*, *Query Id*, and *Database Element*.

V. HPC and Cloud Computing

Cloud computing is a type of computing that relies on sharing computing resources rather than having local servers or personal devices to handle applications. Cloud

computing offers a larger capacity of additional computation and flexibility. However there are some challenges that must be taken into consideration like security and data privacy and moving data back and forth.

On the other hand, HPC requires very low latency and servers with individually high performance. It turns out however, that all MPI workloads are not the same. At the top of the pyramid presented in Figure 3 is with MPI workloads that require a high performance and low latency networks like Infiniband. The middle part of the pyramid is filled with MPI workloads that require a great network, but not an Infiniband network. At the bottom of this pyramid we have the so called embarrassingly parallel (EPP) problems which have no data sharing requirements. In this kind of problems a very large dataset is chopped into pieces, distributed to a large pool of workers, and then the data is brought back and reassembled. It is like a MapReduce functionality. This problem workloads are very commonly run on top of MPI clusters, although some academic institutions build out separate or smaller grids to run them instead.

Cloud can accommodate EPP and HPC workloads, but is not itself necessarily a HPC neither a EPP in the traditional sense. As we explained before, we believe it is a platform for medium-scale HPC applications which are not tightly coupled. The ease of use of HPC applications must be addressed at all layers (infrastructure, platform and software as a service).

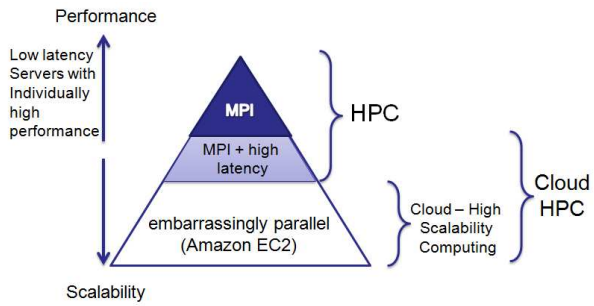

Fig. 3. Parallel algorithm classification.

VI. Conclusions

Large-scale systems considered in our investigations, must be prepared to receive a continuous flow of queries. Therefore, it is not sufficient to accelerate the response time of individual queries, it is necessary to leverage the capabilities of the resources available to effectively respond to as many queries as possible. We presented two paradigms used for parallel processing of queries in metric spaces.

The hybrid implementation was based on overlapping computation with communication through a data partitioning in charge of MPI, with a functional partitioning in each node provided by OpenMP. This involved an coarse-grained OpenMP programming style in each search node. The design and implementation of hybrid system proposed in this paper needed to compensate the different loads on communicating and computing threads through the manipulation of intermediate data structures like input and output *buffers*. These structures allowed the search engines to work independently of receipt and distribution of messages (queries and answers).

In the second proposal, the computational capabilities of many-core GPUs have been exploited to improve the query process of metric spaces. The architecture carefully selects and manages the number of blocks and theirs threads, maximizes the use of shared memory and due to the independence of data, takes advantage of zero cross talk among threads. Additionally, our proposals have zero overload in the data structures, which implies all the available memory can be used for the database.

In order to successfully scale the content-based search to very large numbers of data objects, both proposals have considered the construction of a proper index and the use of an appropiate infrastructure on which it will execute. Though, the performance of the implementations are the subject of the further study, the proposed approaches seem to be promising and might form an alternative to the current state-of-the-art solutions.

VII. Acknowledgements

We wish to thank to the UNSL for allowing us the access to their computational resources. This research has been partially supported by Project UNSL-PROICO-30310, Project UNSL-PROICO-330303 and CONICET.

References

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

[2] E. Chávez, G. Navarro, R. Baeza-Yates, and J. Marroquín, "Searching in metric spaces," *ACM Comput. Surv.*, vol. 33, no. 3, pp. 273–321, 2001.

[3] H. Samet, *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.

[4] K. Figueroa, E. Chávez, G. Navarro, and R. Paredes, "Speeding up spatial approximation search in metric spaces," *ACM Journal of Experimental Algorithmics*, vol. 14, p. article 3.6, 2009.

[5] P. Ciaccia and M. Patella, "Approximate and probabilistic methods," *SIGSPATIAL Special*, vol. 2, no. 2, pp. 16–19, Jul. 2010. [Online]. Available: http://doi.acm.org/10.1145/1862413.1862418

[6] M. Patella and P. Ciaccia, "Approximate similarity search: A multi-faceted problem," *J. Discrete Algorithms*, vol. 7, no. 1, pp. 36–48, 2009.

[7] B. Bustos and G. Navarro, "Probabilistic proximity searching algorithms based on compact partitions," *Discrete Algorithms*, vol. 2, no. 1, pp. 115–134, Mar. 2004. [Online]. Available: http://dx.doi.org/10.1016/S1570-8667(03)00067-4

[8] A. Singh, H. Ferhatosmanoglu, and A. Tosun, "High dimensional reverse nearest neighbor queries," in *The twelfth international conference on Information and knowledge management*, ser. CIKM '03. New York, NY, USA: ACM, 2003, pp. 91–98. [Online]. Available: http://doi.acm.org/10.1145/956863.956882

[9] F. Moreno-Seco, L. Micó, and J. Oncina, "A modification of the laesa algorithm for approximated k-nn classification," *Pattern Recognition Letters*, vol. 24, no. 13, pp. 47 – 53, 2003.

[10] E. Chávez, K. Figueroa, and G. Navarro, "Proximity searching in high dimensional spaces with a proximity preserving order," in *Proc. 4th Mexican International Conference on Artificial Intelligence (MICAI)*, ser. LNAI 3789, 2005, pp. 405–414.

[11] A. Labrinidis and H. V. Jagadish, "Challenges and opportunities with big data," *Proc. VLDB Endow.*, vol. 5, no. 12, pp. 2032–2033, 2012.

[12] G. Hager and G. Wellein, *Introduction to High Performance Computing for Scientists and Engineers*, ser. Computational Science, 2010.

[13] G. Hager, G. Jost, and R. Rabenseifner, "Communication characteristics and hybrid mpio/penmp parallel programming on clusters of multi-core smp nodes," *Proceedings of Cray User Group Conference*, vol. 4, no. d, pp. 54–55, 2009.

[14] D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors, A Hands on Approach*. Elsevier, Morgan Kaufmann, 2010.

[15] J. D. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips, "GPU computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.

[16] NVIDIA, "Nvidia cuda compute unified device architecture, programming guide version 4.2." in *NVIDIA*, 2012.

[17] V. Mancini, F. Bustos, V. G. Costa, and A. M. Printista, "Data partitioning evaluation for multimedia systems in hybrid environments," in *3PGCIC*, 2012, pp. 321–326.

[18] N. R. Brisaboa, A. Fariña, O. Pedreira, and N. Reyes, "Similarity search using sparse pivots for efficient multimedia information retrieval," in *ISM*. IEEE Computer Society, 2006, pp. 881–888.

[19] V. G. Costa, M. Marín, and N. Reyes, "Parallel query processing on distributed clustering indexes," *J. Discrete Algorithms*, vol. 7, no. 1, pp. 3–17, 2009.

[20] V. G. Costa, R. J. Barrientos, M. Marín, and C. Bonacic, "Scheduling metric-space queries processing on multi-core processors," in *PDP*, 2010, pp. 187–194.

[21] R. Barrientos, J. Gomez, C. Tenllado, and M. Prieto, "Heap based k-nearest neighbor search on gpus," 09/2010 2010, pp. 559–566.

[22] V. Garcia, E. Debreuve, and M. Barlaud, "Fast k nearest neighbor search using GPU," in *CVPR Workshop on Computer Vision on GPU (CVGPU)*, Anchorage, Alaska, USA, June 2008.

[23] V. Garcia, E. Debreuve, F. Nielsen, and M. Barlaud, "k-nearest neighbor search: fast GPU-based implementations and application to high-dimensional feature matching," in *IEEE International Conference on Image Processing*, Hong Kong, Sept. 2010.

[24] K. Kato and T. Hosino, "Solving k-nearest neighbor problem on multiple graphics processors," in *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, CCGRID*, ACM, Ed., 2010, pp. 769–773.

[25] Q. Kuang and L. Zhao, "A practical gpu based knn algorithm," in *International Symposium on Computer Science and Computational Technology (ISC-SCT)*, 2009, pp. 151 – 155.

[26] S. Liang, Y. Liu, C. Wang, and L. Jian, "Design and evaluation of a parallel k-nearest neighbor algorithm on CUDA-enabled GPU," in *IEEE 2nd Symposium on Web Society (SWS)*, 2010, pp. 53 – 60.

[27] T.Rozen, K.Boryczko, and W.Alda, "Gpu bucket sort algorithm with applications to nearest-neighbour search," *Journal of WSCG*, vol. 16, no. 1-3, pp. 161–167, 2008.

[28] R. Barrientos, J. Gomez, C. Tenllado, and M. Prieto, "Query processing in metric spaces using gpus," 2011.

[29] R. J. Barrientos, J. Gomez, C. Tenllado, M. Prieto, and M. Marin, "kNN Query Processing in Metric Spaces using GPUs," vol. 6852, 2011, pp. 380–392.

[30] S.Brown and J.Snoeyink, "Gpu nearest neighbors using a minimal kd-tree," in *Second Workshop on Massive Data Algorithmics (MASSIVE 2010)*, Snowbird, Utah, USA, June 2010.

[31] D.Qiu, S.May, and A. Nüchter, "Gpu-accelerated nearest neighbor search for 3d registration," in *Proceedings of the 7th International Conference on Computer Vision Systems: Computer Vision Systems*, ser. ICVS '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 194–203.

[32] K.Zhou, Q.Hou, R.Wang, and B.Guo, "Real-time kd-tree construction on graphics hardware," in *ACM SIGGRAPH Asia 2008 papers*, ser. SIGGRAPH Asia '08. New York, NY, USA: ACM, 2008, pp. 126:1–126:11.

[33] M. Lopresti, N. Miranda, F. Piccoli, and N. Reyes, "Efficient similarity search on multimedia databases," in *XVIII Congreso Argentino de Ciencias de la Computacin, CACIC 2012*, 2012, pp. 1079–1088.

[34] ——, "Permutation index and gpu to efficiently solve many queries," in *Proc. HPCLatam 2013*, 2013. [Online]. Available: http://hpc2013.hpclatam.org/

[35] N. Miranda, E. Chávez, F. Piccoli, and N. Reyes, "(very) fast (all) k-nearest neighbors in metric and non metric spaces without indexing," in *Proc. International Conference on Similarity Search and Applications (SISAP 2013)*. A Coruña, Spain: Elseiver, 2013.