WCCI 2012 IEEE World Congress on Computational Intelligence
June, 10-15, 2012 - Brisbane, Australia IJCNN

Parallel Algorithms for Tensor Product-based Inexact Graph Matching

Lorenzo Livi, Antonello Rizzi
Department of Information Engineering, Electronics and Telecommunications
SAPIENZA University of Rome
Via Eudossiana 18, 00184, Rome, Italy
livi@diet.uniroma1.it, antonello.rizzi@uniroma1.it

***Abstract*—In this paper we face the inexact graph matching problem from the parallel algorithms viewpoint. After a brief introduction of both graph matching and parallel computing contexts, we discuss a specific method of performing inexact graph matching based on the well known tensor product operator. We analyze the problem using two parallel computing models, following different algorithmic strategies, and performing also an experimental evaluation. The aim of this paper is to provide modeling and algorithmic strategies to extend inexact graph matching methods to graphs of high order and size, conceiving the computational problem in the more wider context of graph-based Pattern Recognition and Soft Computing systems. As a whole, the obtained results encourage more effort on this direction.**

I. Introduction

A great number of interesting Pattern Recognition problems coming from real world applications must cope with structured and relations-based patterns, such as chemical compounds, smart grids, computing and social networks, for instance. As a result, each pattern can be represented as a *labeled graph*, where vertices and edges are possibly equipped with complex labels, able to encode different kind of information. For this purpose, developing recognition systems able to cope with labeled graphs is a fundamental step. Consequently, the field of *graph-based Pattern Recognition* is growing fast, aimed at the establishment of effective and efficient Pattern Recognition techniques on the domain of graphs. A key procedure in these systems is the well known *inexact graph matching*, conceived as a (dis)similarity measure on the input set of labeled graphs [1], [2], [3], [4]. A problem of current and future interest is the computational speed-up of these methods. Indeed, being the basic building block of the recognition system, inexact graph matching procedures are usually employed in computational intensive tasks, such as classification model and algorithm parameters optimization for the problem at hand. Consequently, looking at the recognition system as a whole, the computational demand becomes, in practice, quickly very high, posing some limitations to the applicability of these systems.

The Random Access Machine (RAM) computational model is going to reach a *bottleneck* considering the algorithmic demand for large datasets and the hardware design trend. The huge growth of the on-chip complexity has induced different theoretical computing branches, namely, from the complexity theory to algorithms design. The first one focuses on the identification of computational problems that are *intimately serials*, aiming at their theoretical classification. The second one is motivated by the need of a well established mathematical framework for parallel algorithms design [5]. Unfortunately, still now there is not a widely accepted *bridging model* for parallel computation, mostly due to the multitude of quickly growing parallel hardware. A particularly interesting, and relatively cheap, (physical) computing model, considering both hardware and software perspective, is the one that can be referred as *hybrid* multicore CPU–GPU computing, where the standard multicore processors and the Graphic Processing Units (GPUs) [6] are used together to accomplish some computational intensive task. Nonetheless, multicore CPU and GPU computing systems are very interesting and effective also when used separately.

In this paper, our interest is focused on dealing with the inexact graph matching problem using the framework provided by parallel algorithms. In particular, we focus on speeding up the computation of the *tensor product* operation between labeled graphs [7], conceived as the most computationally critical part of certain types of graph-based Pattern Recognition and Soft Computing systems. In the following, we will describe the computational problem using different abstract models, defining different algorithmic strategies based on these models. We will also give an experimental evaluation of the effective time performance using two different parallel computation architectures, that is, multicore CPU and GPU.

This paper is organized as follows. In Section II we introduce the Parallel Computing context, together with a brief overview of two abstract computation models. In Section III we introduce the inexact graph matching problem, detailing, in Section III-A, the methods based on the tensor product operator. We report two different analysis for a specific and a more general scenario, described in Section III-B1 and III-C, respectively. In Section IV we report an experimental analysis of the proposed methods. Finally, in Section V we draw our conclusions.

II. Parallel Computing

In the literature, various mathematical models of parallel machines have been proposed [8]. For instance, we can cite *Boolean circuits* [9], *Parallel Random Access Machines*

U.S. Government work not protected by U.S. copyright

(PRAMs) [10], [11] and different network-based systems, such as *Array*, *Hypercubes* and *Grids* [12]. Roughly speaking, these models differ in the level of *granularity* adopted for the description of the computation and, consequently, in the model parameters. Another fundamental aspect concerns the *communication* and *synchronization* issues between the *processing units* involved in the parallel computation. It is possible to distinguish two types of communication schemes for parallel machines: *shared memory* based, called multiprocessors architectures, and the *network* based, where different processing units communicate through a network (e.g., a point to point network).

The *parallel time*, denoted with $t(n)$, is the time needed to complete the execution of the parallel algorithm, assuming an input of size n. The notation $p(n)$ stands for the number of processing units, or simply processors, involved in the computation, defined as a function of the input size. The *cost* of a parallel algorithm is defined as $C(n) = t(n) \cdot p(n)$. A problem is said to be solvable efficiently by a parallel algorithm if its parallel time is polylogarithmic, $t(n) = \log(n)^{O(1)}$, using a polynomial number of processors, $p(n) = n^{O(1)}$. Problems of this kind belong to the $\mathbf{NC}$ complexity class [5]. An important measure of performance comparison can be established between a parallel solution of a given problem instance and the relative efficient known serial solution. The *speed-up* is a measure to determine the gain achieved with the parallel version of an *optimal* serial algorithm for a feasible problem, and is defined as $s(n)/t(n)$, where $s(n)$ is the worst-case optimal serial time known for the given problem. The ideal speed-up is equal to the number of employed processing units $p(n)$, meaning that the parallel version is exactly $p(n)$ times faster than the optimal serial one. When this result is achieved, we have an *optimal parallel algorithm* for the given problem.

A. PRAM Model

The Parallel Random Access Machine (PRAM) model [10], [11] is the natural multiprocessor extension of the original serial RAM model. Indeed the basic version of PRAM consists in an (unbounded) number of RAM processing units $P_0, P_1, \ldots$ and a collection of *shared memory cells* $C_0, C_1, \ldots$. Each processor has its own local memory. The communication between these processing units is carried out via (concurrent) accesses to the shared memory. The PRAM's main computation cycle is a synchronized three-stage cycle, where firstly the read operations are done on the shared memory, then follows the individual local computations, and finally the write operations are performed toward, again, the shared memory. Each shared memory access (read or write) is assumed to have a constant time cost. In this model, there are single shared memory cell synchronization issues, that are managed in four different ways:

- CRCW: the Concurrent-Read Concurrent-Write PRAM is able to correctly manage concurrent read/write accesses to the same shared memory cell.
- CREW: the Concurrent-Read Exclusive-Write PRAM is able to consistently manage only concurrent reads.
- CROW: the Concurrent-Read Owner-Write PRAM is similar to CREW, but permits only to the pre-assigned owner of the cell the write operation.
- EREW: the Exclusive-Read Exclusive-Write PRAM has the weakest concurrency features, because no concurrent operation is allowed.

The CRCW PRAM is certainly the most powerful example of PRAM, but it is possible to show that these models are actually equivalent [5]. An important characteristic of PRAM based algorithms is their *transportability*, within a constant cost factor, between different synchronization schemes, and different number of available processing units p.

Unfortunately, an unbounded number of processors is not physically possible, so in practice any PRAM implementation is bounded by say p processors, normally expressible as a function of the input size n, $p = n^{1-\alpha}, 0 \leq \alpha \leq 1$. However, usually $p \ll n$, and then the achievable gain is *only* of a constant factor of the original serial optimal complexity.

B. BSP Model

The Bulk Synchronous Parallel (BSP) model [13] is aimed at the establishment of a bridging computational model to develop algorithms (and then software) for different parallel hardware, where the architecture is composed by different processing units that are *locally independent* and for which is required, and explicitly modeled, a communication and synchronization cost. Indeed the BSP model aims to introduce the concept of *portable parallel algorithms*, that are parameter-aware algorithms that run efficiently on different machines with the widest range of values of these parameters. To achieve this generality, in the original formulation, a compromise is reached without taking care of the *locality* of the computation. Another important detail is that in the BSP model, the memories size are not taken explicitly into account.

Formally, the BSP model is a triple (v, g, l), where v is the number of *virtual* processing units, each equipped with a local memory to be used in the local computation (usually $p \leq v$). The parameter g, called the *router*, expresses the cost of a single-word *continuous* point-to-point communication among the processing units. That is, the parameter g should be understood as the throughput of the communication system. Finally, the parameter l models the cost for a global and periodic *barrier synchronization*. That is, l is the minimum number of local *time steps* (local basic operations) performed during two successive synchronizations. The BSP computation is considered as a global synchronized *superstep*, that consists in v local independent computations, a messages exchange and finally a global barrier synchronization, that also contributes to prevent deadlocks and livelocks. To model the communication cost as a whole, the notion of *h* relation is introduced. The value *h* represents the maximum number of inbound/outbound messages at a given processing units. If the computation network is modeled as a graph, this is the maximum in/out degree of the graph. The standard cost model [13] of a single superstep is then expressed as

$$\max_i w_i + \max_i h_i g + l \tag{1}$$

where w_i is the serial computation cost for the *i*-th processing unit. Usually, the maximum values are implicitly included in Equation 1, simplifying the equation as $w+hg+l$. The cost of a given BSP program is just expressed as the sum of these supersteps costs. Note that in practice, the values of g and l are determined empirically, estimating the system at hand. Hence, the BSP model does not falls purely into the shared memory-based types of machines.

The original BSP model has been extended in many ways [14], [15], [16]. Recently, the Multi-BSP model [16] was proposed, introducing a *hierarchical* representation of the computation and an explicit parameter for the available memory size. The hierarchy is introduced to better model the physical reality of the parallel system, that is, different memories and processors capabilities at each level of observation.

III. Inexact Graph Matching

The graph matching problem is of utmost importance in many graph-based recognition systems, and is characterized by both theoretical and practical issues. The numerous matching procedures proposed in the technical literature can be classified into two well defined families, those of *exact* and *inexact matching*. The first one is just an *equivalence* test, while the latter is a more complex and interesting problem where the challenge is in computing *how much* two given labeled graphs differ. These measures, to be fully meaningful, must take into account both topological and labels related information. However, it is known that the current great obstacle in this field is their computational demand. As a basic building block of more complex Pattern Recognition systems, these methods should be very efficient, to be able to effectively preserve, for example, the algorithm adaptability to a specific dataset via parameters optimization in a reasonable computing time. Moreover, the efficiency is required if we deal with large datasets or, even more, with big labeled graphs. Note that big graphs are encountered often in many fields of high interest, such as social networks, bio-chemical compounds, proteins–proteins interaction networks, smart grids and so on. In the current scientific literature it is possible to distinguish three mainstream approaches: *Graph Edit Distance* based [17], [1], *Graph Kernels* based [2], [3], [18], [19] and *Graph Embedding* based [4], [20], [21].

A. Tensor Product of Graphs

The topology of a graph $G = (\mathcal{V}, \mathcal{E}), |\mathcal{V}| = n$, can be fully represented by its adjacency matrix $\mathbf{A}^{n\times n}$. Anyways, when graphs are employed to model patterns, they are labeled usually with complex information (e.g., feature vectors, composite types, etc.). Consequently, the full description of such a graph is achieved using, in addition to the adjacency matrix $\mathbf{A}$, vertices and edges *labeling functions*. This generalization is obtained introducing the labeled graph. A labeled graph is a tuple $G = (\mathcal{V}, \mathcal{E}, \mu, \nu)$, where $\mathcal{V}$ is the (finite) set of vertices (also referred as nodes), $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is the set of edges, $\mu : \mathcal{V} \rightarrow \mathcal{L}_{\mathcal{V}}$ is the vertex labeling function with $\mathcal{L}_{\mathcal{V}}$ the vertex-labels set, and $\nu : \mathcal{E} \rightarrow \mathcal{L}_{\mathcal{E}}$ is the edge labeling function with $\mathcal{L}_{\mathcal{E}}$ the edge-labels set. Such a graph is able to model a broad range of patterns.

The tensor product of graphs (also called *Direct Product*) [7] is founded on the Kronecker product between matrices of arbitrary sizes.

Definition 3.1 (Kronecker Product): Given two real matrices $\mathbf{A}^{n\times m}$ and $\mathbf{B}^{p\times q}$, the Kronecker product is denoted $\mathbf{A} \otimes \mathbf{B} = \mathbf{W} \in \mathbb{R}^{np\times mq}$, and is defined as

$$\mathbf{W} = \begin{bmatrix} A_{1,1}\mathbf{B} & A_{1,2}\mathbf{B} & \cdots & A_{1,m}\mathbf{B} \\ \vdots & \vdots & \vdots & \vdots \\ A_{n,1}\mathbf{B} & A_{n,2}\mathbf{B} & \cdots & A_{n,m}\mathbf{B} \end{bmatrix}$$

Deeply looking at the operator shown in Definition 3.1, it is easy to understand that it is defined as a bulk of totally independent computations. Let G_1 and G_2 be two labeled graphs, with, for notational purpose, $\mathcal{V}_1 = \{v_1, ..., v_n\}$ and $\mathcal{V}_2 = \{u_1, ..., u_m\}$. The tensor product $G_1 \otimes G_2$ produce a graph, called tensor product graph $G_\times = (\mathcal{V}_\times, \mathcal{E}_\times)$, with $\mathcal{V}_\times = \mathcal{V}_1 \times \mathcal{V}_2$, and $\mathcal{E}_\times = \{((v_i, u_r), (v_j, u_s))|\ (v_i, v_j) \in \mathcal{E}_1, (u_r, u_s) \in \mathcal{E}_2\}$. In Figure 1 it is shown an illustrative example.

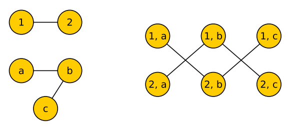

Figure 1: Tensor Product between Graphs

It is possible to show that performing a random walk on the tensor product graph $G_\times$ is equivalent to performing a simultaneous random walk on G_1 and G_2. There are many other important properties derived from the generalized product between graphs [7], [22], [23]. Therefore, the tensor product is a very interesting candidate in the inexact graph matching context, especially in the graph kernels family [2], [19], [24]. For the purpose of Pattern Recognition problems, the basic tensor product formulation has been adapted taking into account also the labels of both vertices and edges. The resulting *weighted adjacency matrix* $\mathbf{W}$ of the tensor product graph $G_\times$ is defined on the base of a *valid composite kernel function* $k(\cdot,\cdot)$, defined as a product of three different valid kernel functions, able to deal with vertex and edge labels peculiarities. Assuming an edge in $G_\times$ between (v_i, u_r) and (v_j, u_s), it is possible to use different specialized kernels $k_{\mathcal{V}}(\cdot,\cdot), k_{\mathcal{E}}(\cdot,\cdot)$ as follows

$$\begin{aligned} W_{(v_i,u_r),(v_j,u_s)} &= k((v_i,u_r),(v_j,u_s)) = \\ &= k_{\mathcal{V}}(v_i,u_r) \cdot k_{\mathcal{E}}((v_i,v_j),(u_r,u_s)) \cdot k_{\mathcal{V}}(v_j,u_s) \end{aligned} \tag{2}$$

Conversely, if the edge does not exists, a value of zero is assigned. The basic valid kernel functions $k_{\mathcal{V}}(\cdot,\cdot)$ and $k_{\mathcal{E}}(\cdot,\cdot)$ can be evaluated as Gaussian RBF kernels, for instance. Note that different valid composition rules can be conceived in Equation 2. The resulting graph $G_\times$ is known to be able to *encode* the similarity of the two input graphs. This fact can be exploited in many different ways, extracting and elaborating this information from $\mathbf{W}$ for recognition and learning purposes.

In the remainder of this paper, where not explicitly stated otherwise, we assume always to have to deal with two labeled graphs G_1 and G_2 of order n and m, respectively, with $n > m$, without losing in generality.

1) Serial Computational Complexity: The computational complexity of the optimal serial computation of the tensor product between two graphs G_1 and G_2 is of the order $O(n^2m^2)$. Actually we match labeled graphs, so the real complexity should take into account the cost of computing the various labels (dis)similarity. Assuming a constant cost d for each kernel evaluation (Equation 2), the computational complexity becomes $O(d \cdot n^2m^2)$. We choose to add also the constant factor d because the nature of the labels set $\mathcal{L}_\mathcal{V}$ and $\mathcal{L}_\mathcal{E}$ can be complex, such as text excerpts, sequences of objects, chemical formulas and so on. Consequently, in a finite range analysis of the algorithm, these computations may afflict seriously the total efficiency. We note also that if the input graphs are undirected, the relative adjacency matrices are symmetric. Therefore, only the lower (upper) diagonal of the adjacency matrices should be considered in the tensor product operator ($A_{ij}\mathbf{B} = A_{ji}\mathbf{B}$, see Definition 3.1), reducing the computational complexity by a valuable constant factor. However, the same observation is not always true if we compute the tensor product using the *transition matrices* of the graphs, because the symmetry property is not assured.

B. Graph Coverage Similarity Measure

The graph coverage [3] is a simple, yet interesting, graph kernel function based on the analysis of the relative *weight* of the tensor product graph $G_\times$, denoted with $W(G_\times)$, against the *optimal* one, achievable for a specific pair of graphs. It is basically founded on the well known format provided by the tensor product, described in Section III-A. Given two input arbitrarily labeled graphs G_1 and G_2, let $G_\times^{(1,2)} = G_1 \otimes G_2$, $G_\times^{(1,1)} = G_1 \otimes G_1$ and $G_\times^{(2,2)} = G_2 \otimes G_2$ be their respective possible tensor product graphs. The coverage of G_1 and G_2 is defined as

$$\kappa(G_1, G_2) = \frac{W(G_\times^{(1,2)})}{\max\{W(G_\times^{(1,1)}), W(G_\times^{(2,2)})\}} \qquad (3)$$

with $W(G_\times^{(1,2)}) = \sum_{i=1,j=1}^{|\mathcal{V}_\times|} W_{ij}$, where, in this case, $\mathbf{W}$ is the weighted adjacency matrix of $G_\times^{(1,2)}$.

In [3, Figure 1] are shown the time performance of the graph coverage computation. As it is easy to understand, the required computing time becomes impractical even with graphs of modest order and size.

An interesting observation is that the graph coverage, in contrast to different (random) walk based graph matching methods founded on the tensor product computation [2], [19], [24], does not requires the explicit determination (and thus allocation) of the weighted adjacency matrix $\mathbf{W}$ of the tensor product graphs. Indeed, it is needed only the sum of the components of $\mathbf{W}$. This fact has a huge impact on the memory consumption, and consequently, also for the general computational efficiency. Therefore, the graph coverage algorithm, at least from the memory usage viewpoint, is an interesting candidate for the application of inexact graph matching methods to very large graphs.

1) Parallel Graph Coverage: From Equation 3, it is easy to understand that the computation of the graph coverage requires three *independent* tensor product computations. Consequently, one straightforward approach is to compute these three products in parallel. The serial computational complexity of the graph coverage is given by $O(d \cdot 3n^4)$. With this approach, we obtain a theoretical speed-up of a constant factor 3, that is $t(n) = O(d \cdot n^4)$. As we will see in Section IV-A, this simple approach yields a practical relevant speed-up. However, with this strategy we are not able to improve the performance as the number of available processors p increases.

C. Parallel Tensor Product Computation

A more general and effective parallel computation strategy can be conceived for all the inexact graph matching methods based on the tensor product computation. A flexible parallel algorithm should be able to scale as the number of available processors p increases. However, the results of the analysis itself are dependent on the specific adopted model. For this purpose, we discuss and analyze two specific parallel architectures, namely a general multicore system and a hybrid two devices computation, using PRAM and BSP models, respectively.

1) Multicore Computation Using the PRAM Model: A multicore system is a single shared memory multiprocessor architecture (e.g. multicore CPU or GPU), where communication and synchronization costs among the involved cores can be considered negligible for algorithmic purposes. Thus, we can model the multicore system using a PRAM CREW, assuming a total number $p \leq n$ of available processing units. We can assign to each processor an independent block of n^2/p elements (actually 0-1 values) of the adjacency matrix of G_1, that is $\mathbf{A}_1$. Each element a_{ij} of this block is independently multiplied with the adjacency matrix of G_2, that is $\mathbf{A}_2$, computing a specific $m \times m$ block of the resulting matrix $\mathbf{W}$. Consequently, the parallel time for the computation of $\mathbf{W}$ as a whole will be $O(d \cdot n^2m^2/p)$, where d is again the cost for each labels dissimilarity computation. If $p(n) = O(n)$, then we have a parallel time of $O(d \cdot nm^2)$, achieving an optimal speed-up. If this simple abstract model is applied to a GPU device, the concurrent read capability (CR) may not be fully verified and cost-free in practice. The concurrent reading of labels and adjacency matrices related data can become a problem. Indeed, there are huge access speed differences among the various memories available in the GPU device, with additional bidirectional CPU–GPU data transfer costs [6], [25]. So, in this scenario we expect to obtain a real speed-up consistently lower than p. In fact, in [26] the GPU was modeled using the BSP, taking directly into consideration the synchronization and communication costs.

2) Hybrid Computation Using the Standard BSP Model: A more powerful computation strategy would be the one that conceives the concurrent usage of two independent locally-connected devices, such as for example the multicore CPU and the GPU, or multicore CPU and a *Field Programmable Gate Array* (FPGA) [27]. A GPU, roughly speaking, is a grid-like interconnection of *streaming multiprocessors*, each equipped with different cores. Usually, a GPU contains from tens to hundreds cores, and is able to run and manage many concurrent threads following the *Single Instruction Multiple Threads* (SIMT) computing paradigm. For tasks like matrix multiplication, the GPUs are known to be very effective devices. However, modeling this hybrid system (multicore CPU and GPU) as a whole is a non-trivial task. Indeed, we think that it is mandatory the adoption of a computation model able to describe also the synchronization and communication costs.

In this scenario, we adopt the standard BSP model, assuming a total number p of available processing units, without discriminating between the ones of the multicore CPU and the ones belonging to the GPU. Due to the computation nature of the tensor product (see Definition 3.1), we assign to each processor, also in this scenario, a block of $n^2/p \cdot m^2$ independent local computations. For what concerns the communication cost we can observe that the number of elements to exchange is composed of n^2/p scalar elements of the first adjacency matrix, and the whole m^2 elements of the second graph. Moreover, also the labels related information of both graphs must be shared. Let $\mathbf{U}^{(k)}$ be the *labels matrix* related to a labeled graph G_k, defined as $U_{ij}^{(k)} = \nu(e_{ij})$, and $U_{ii}^{(k)} = \mu(v_i)$. For the purpose of this paper, we can assume, for consistency, that is $\mathcal{L}_{\mathcal{V}} = \mathcal{L}_{\mathcal{E}} = \mathbb{R}^d$, so the labels matrix $\mathbf{U}^{(k)}$ is well defined. If the graph G_k is of order n and size $s = |\mathcal{E}|$, the matrix $\mathbf{U}^{(k)}$ can be seen as $r_k = (2s + n) \cdot d$ scalar elements. Thus, the number of incoming messages, for each processor, is given by $(n^2 + r_1)/p + (m^2 + r_2)$. In this scheme there will be only one superstep; consequently the synchronization cost is equal to the system synchronization cost, that is l. The BSP cost of this algorithm is given by

$$\left(\frac{d \cdot n^2 m^2}{p}\right) + \left(\frac{n^2 + r_1}{p} + m^2 + r_2\right) g + l \tag{4}$$

This cost asymptotically converges to the optimal parallel one $O(d \cdot n^2 m^2/p)$ only if $g = O(d \cdot m^2)$ and $l = O(d \cdot n^2 m^2/p)$. The throughput parameter g can be estimated practically on the base of the particular PCI-express communication bus capacity (transfer rate) available on the system at hand.

IV. Experimental Evaluation

We report different experimental evaluations based on the parallel algorithms analysis provided in Section III, and relative subsections. The stress tests are conceived to understand the feasibility and extendability of these methods to graphs of higher order and size, and in the same time to verify experimentally the expected improvements. The aim is to achieve a considerable speed-up for the purpose of Computational Intelligence techniques in graph-based Pattern Recognition and Soft Computing systems, such as Clustering-based classifiers with automatic kernel functions parameters tuning (for instance through evolutionary optimization algorithms), or even simpler ones, such as graphs classifiers based on the k-NN rule. Thus, the reported computing time performances, and relative observations, must be contextualized in these scenarios.

The sample graphs have been generated using *Markov Chains*, and both vertices and edges of all graphs have been labeled always with randomly determined 10-dimensional real vectors. We evaluate *dense* simple graphs of order n from 10 to 300, with a size $s \simeq n \cdot (n-1)/4$, due to the stochastic nature of the generation process. For the purpose of this paper, we do not consider that the input graphs are undirected, analyzing the full adjacency matrices of both graphs. The following tests of Sections IV-A and IV-B have been executed on a machine with an Intel Core 2 Quad CPU Q6600 2.40GHz and 4 Gb of main memory, running a 64-bit Linux OS. The computing time (always expressed in seconds in the paper) is measured using the *clock()* function from the *ctime* library, and takes into account, for each setup of the samples, the average of three graph matching computing times.

However, we stress that the purpose of this paper is to show that parallel algorithms can be employed to speed-up significantly the computation, and not to benchmark a specific graph (dis)similarity algorithm against other state of the art methods or between different particular implementations. Indeed, the effective performance outcome depends on many important details, primarily the adopted programming language.

A. Specific Graph Coverage Parallelization

In this section, we report the experimental evaluation of the analysis proposed in Section III-B1, concerning the direct graph coverage parallelization. As it was expected, it is possible to see in the charts plotted in Figure 2 that the parallel version of the graph coverage (in the figure, *Parallel GC*) is, with a tight approximation, three times faster than the serial one (in the figure, *Serial GC*). Indeed the parallel computation of the three weights of the respective tensor product graphs can be accomplished totally in parallel. However, there is an implicit limitation induced by the fact that this approach does not scale as the available processing units increase, reducing the interest for this strategy.

B. Multicore Tensor Product Computation

As we have observed in Section III-C1, the direct parallel computation of the tensor product, using p processing units, permits to achieve a scalable optimal speed-up of a factor p. For this purpose, we have implemented a multicore version of the tensor product operator, using a configurable number

p of processing units (i.e., cores). To be able to better analyze the results, we have used, as graph similarity measure, again the graph coverage scheme. That is, we employ this parallel version of the tensor product into the graph coverage (dis)similarity evaluation scheme (see Equation 3), computing each tensor product using the parallel algorithm given in Section III-C1, and reporting only the computing time as a whole.

Due to the limited parallel capability of the available hardware, we have used $p = 4$. In Figure 2 are shown the results for the multicore implementation (in the plot denoted as *Parallel TP 4c*). It is easy to understand that the analysis described in Section III-C1 finds confirmation through this test. Indeed, we obtain, with tight approximation, a speed-up factor of 4. Note that the speed-up tends to be equal to p as the order (and in particular the average size) of the graphs grows. This intuitive result is due to the constant costs associated with threads management, that become negligible when the computation become heavier. A selected sampling of these results is shown in Table I.

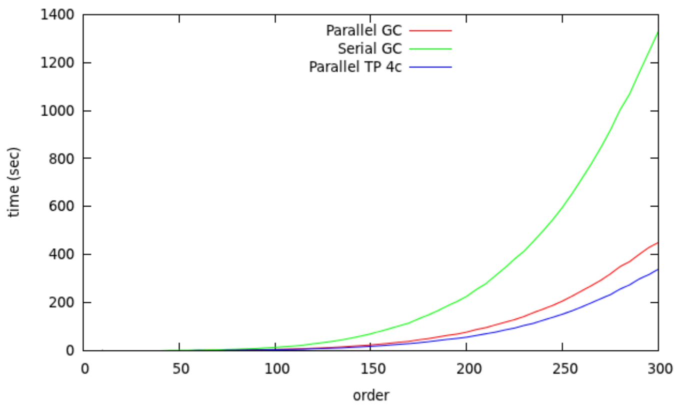

Figure 2: Performance Results for the CPU

Order	Avg. Size	Serial GC	Parallel GC	Parallel TP 4c
10	19	$\simeq 0$	$\simeq 0$	0.001
30	140	0.093	0.032	0.036
55	461	1.013	0.344	0.271
100	1622	12.843	4.302	3.275
150	3712	69.800	23.313	17.450
200	6566	225.507	76.671	55.510
300	14832	1330.820	450.154	339.039

Table I: Sampling of Time Performance of Figure 2

C. Parallel Tensor Product Computation using the GPU

GPU devices provide large scale multicore capability in a relatively easy to use format. For example, the *NVIDIA CUDA* framework [28] is a low level set of *C* routines to interact with a CUDA-enabled GPU device. For the purpose of the tensor product computation using a single multiprocessor device (see Section III-C1), the adoption of a GPU provides a simple, yet effective, way to augment the number of available processing units p. However, the GPU management has an implicit cost, that can be summarized in memory allocation and CPU–GPU data transfer costs. In this context, the GPU is also referred as the *device* and the CPU as the *host*. Our available GPU is a *GeForce GT 240* equipped with 1 GB of global memory. It consists of 12 multiprocessors, with 8 cores each, for a total of $p = 96$ cores. The estimated *PCI-Express x16 Gen1* bandwidth for host–device and device–host communications are 2142.7 MB/sec and 1411.3 MB/sec, respectively. Our Linux-based *CUDA* runtime, driver and capability versions are 3.2, 4.0 and 1.2, respectively.

GPU programming requires a different thinking about the program execution. Indeed the GPU performs a *vectorized* access to the memory issuing what is called a *warp* of threads. Roughly speaking, when using a GPU it is mandatory to maximize the parallel and independent access of the running threads to the data [28]. For our purposes, this translates in maximizing the parallel access of the threads to the two input adjacency matrices, computing, again independently, the resulting matrix $\mathbf{W}$. Therefore, the computation of the tensor product is demanded to the GPU, with the so called *GPU kernel* routines, that are *C* functions that run on the GPU hardware. For misunderstanding avoidance with the valid kernel functions of Equation 2, in the remainder of this paper we will call these *C* functions simply GPU functions or device functions. Our GPU function for the tensor product computation uses a 2D grid of *blocks*, each defined by different threads. In this strategy, the matrix $\mathbf{W}$ is explicitly computed and sent back to the host, eventually computing the weight. Considering the simple processing engines of the GPU cores [28], the dissimilarities between the graphs labels are assumed to be pre-computed and available. Indeed, delegating to the GPU also the computation of each valid kernel function (i.e., $k_{\mathcal{V}}(\cdot,\cdot)$ and $k_{\mathcal{E}}(\cdot,\cdot)$ of Equation 2), would certainly yield a total loss of performance in this scenario. For what concerns the timing routines, the GPU functions execution times have been measured using the CUDA *Events* functions [28].

Algorithm 1 shows a sketch of the strategy used to compute a single tensor product between two labeled graphs. Listing 1 shows the code of the main GPU function using a 2D grid of threads. Note that, in this implementation, the matrices are represented as vectors with N components (i.e., N is the total number of components in the matrices). The device function $labels(\cdot)$ accesses the pre-computed labels dissimilarities. Replicating the experiments of Sections IV-A and IV-B, we perform again the test in the graph coverage computation scenario. In Figure 3 are shown the results of this test, together with the already-shown ones in the previous section. Moreover, in Table II we report a selected sampling of these results. As it was expected, we start to observe a good gain in using the GPU from graphs of order about 70. This is due to the implicit GPU function setup costs, that become relatively tolerable as the task becomes harder. It is possible to observe that with graphs of order 120, and average size 2384, we obtain a speed-up factor of roughly 9 against the serial computation. As noted in Section III-C1, the achievable speed-up using a GPU with $p = 96$ cores, cannot be around 96 times for any computation.

However, this strategy explicitly allocates the matrix $\mathbf{W}$ on the GPU global memory. As it is easy to understand, the matrix $\mathbf{W}$ become quickly very big. For example, with graphs of order 100, and using single-precision floating point representation for the numbers, the memory requirement is of $\simeq 381$ MB, and with graphs of order 120 is $\simeq 791$ MB. Indeed, with the available hardware, and without using *pinned* memory techniques [28], we are unable to process graphs of order greater than 120.

In Table III we report some details concerning the results obtained with the GPU. We show the total computing time of the program as a whole (Total), the sum of the three GPU functions execution times for the computation of the matrix $\mathbf{W}$ (GPU Func.), the difference between these two quantities (Setup), which basically represents memory allocation and CPU–GPU data transfer time-related costs, and finally the relative percentage of time spent doing GPU setup against the total computing time (% Setup/Total). It is worth noting that there is a huge difference between the relative management time needed for graphs of order 20 and 120. Indeed, we need to allocate and transfer bigger matrices, and consequently perform more operations, but the ratio of the time spent only in setup activities drops significantly.

Algorithm 1 Sketch of GPU-based Tensor Product Computation

```
1: Allocate, on the host and device memory, matrices $\mathbf{A}_1, \mathbf{A}_2, \mathbf{W}$.
2: Allocate on the device constant memory the vertices and edges labels.
3: Transfer $\mathbf{A}_1, \mathbf{A}_2$ to the device.
4: Execute the GPU function for the tensor product computation.
5: Transfer the computed matrix $\mathbf{W}$ back to the host memory.
6: Compute the weight of $\mathbf{W}$ on the host.
```

Listing 1: 2D Main GPU Function

```
__global__ void tensorProduct2D(float* W, int* A1, int* A2)
{
    //X index on the grid
    int ix = blockIdx.x * blockDim.x + threadIdx.x;
    //Y index on the grid
    int iy = blockIdx.y * blockDim.y + threadIdx.y;
    //Position in the output matrix
    int idx = iy * N + ix;

    if(A1[idx/N])
        W[idx] = A1[idx/N] * A2[idx%N] * labels(idx);
}
```

Order	Serial GC	Parallel GC	Parallel TP 4c	GPU
10	$\simeq 0$	$\simeq 0$	0.001	$\simeq 0.001$
30	0.093	0.032	0.036	0.020
55	1.013	0.344	0.271	0.178
100	12.843	4.302	3.275	1.629
120	28.153	9.421	7.071	3.223

Table II: Sampling of Time Performance shown in Figure 3

1) Alternative GPU Computation Scheme: We performed the same test following another GPU approach, that is computing directly the weight on the GPU, without explicitly allocating and transferring the matrix $\mathbf{W}$. The computation of the weight rely on a synchronized access scheme to a shared variable performed directly by the GPU hardware, called *atomic operations* [28]. As it is intuitive to understand, with this strategy, we get rid of the costs related to the management of $\mathbf{W}$, but we introduce synchronization costs related to the concurrent computation of the weight by the running threads of the GPU. Indeed with this strategy the performance gets worse. As an example, with graphs of order 55 the total computing time is 12.093 seconds, that is, roughly 40 times slower. However, we think that there is room for many other experimentations and improvements in this scenario, since the GPU provides many alternatives and programming techniques.

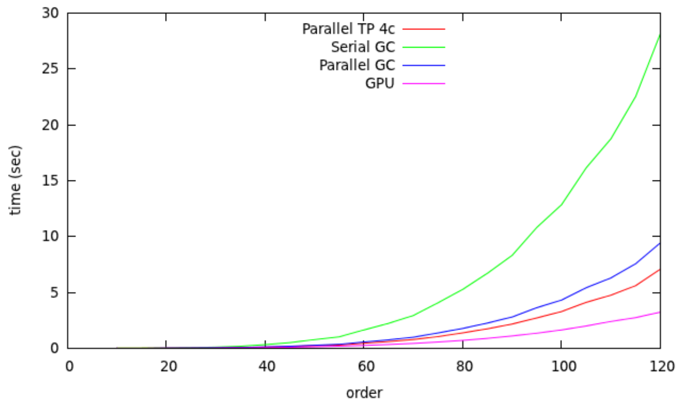

Figure 3: Tensor Product Results using the GPU

Order	Total	GPU Func.	Setup	% Setup/Total
20	0.071	0.0049642	0.0660358	93.00
35	0.099	0.0338106	0.0651894	65.84
60	0.310	0.2394778	0.0705222	22.74
100	1.730	1.6296977	0.1003023	5.797
120	3.372	3.2239041	0.1480959	4.391

Table III: Details of GPU Performances

V. Conclusions and Future Directions

In this paper, we have applied the formal framework of parallel algorithms to the computation of tensor product-based inexact graph matching procedures. The aim was to establish a groundwork for different experimental evaluations based on abstract machine models. Moreover, we have experimentally shown that it is possible to obtain, even when dealing with dense labeled graphs, a considerable, and scalable, speed-up. The obtained results are encouraging, general enough and useful for the purpose of many Soft Computing and Pattern Recognition systems; however, it could be interesting to evaluate the same data on different and more powerful architectures, such as better-equipped GPU devices, programmed with more advanced techniques, and larger multicore systems.

One of the future goal will be the experimental evaluation of the analysis proposed in Section III-C2, using the hybrid computation system provided by the concurrent usage of the multicore CPU and GPU. We think that this kind of hybrid system provides a cheep way to scale significantly with the processing units involved in the computation, hopefully, yielding very good performance for larger graphs. Another valuable improvement is the design of a GPU-based strategy, able to compute directly, and efficiently, the outcome of the valid kernel functions of Equation 2. Moreover, from the abstract modeling viewpoint, it can be of interest the adoption of a multilevel abstract model, such as the Multi-BSP briefly introduced in Section II-B, employing, as lower level entities of this hierarchical representation, two dedicated PRAM machines, one for the multicore CPU and one for the GPU device. Another interesting alternative is the one provided by the development of an *ad-hoc* device, such as a dedicated FPGA circuit implementation for the tensor product computation. For what concerns the analysis of very large patterns, such as dense graphs of order 10^3 and 10^4, it seems necessary to shift the focus on *distributed computing* architectures, where memory and computing capabilities are far beyond the possibility of today parallel computing systems.

In addition, it is worth to study the speed-up of inexact graph matching algorithms from the viewpoint of different approaches, such as Graph Edit Distance and Graph Embedding-based methods. Indeed, there exist matching algorithms with lower theoretical serial computational complexity [4], [1], [18]. However, it is always important to analyze the real performance of algorithms on shared benchmarking datasets, because usually there are strategy-dependent factors that influence significantly the effective achievable speed-up.

References

[1] K. Riesen and H. Bunke, "Approximate graph edit distance computation by means of bipartite graph matching," *Image Vision Comput.*, vol. 27, pp. 950–959, June 2009.

[2] K. M. Borgwardt, C. S. Ong, S. Schönauer, S. V. N. Vishwanathan, A. J. Smola, and H.-P. Kriegel, "Protein function prediction via graph kernels," *Bioinformatics*, vol. 21, pp. 47–56, January 2005.

[3] L. Livi, G. Del Vescovo, and A. Rizzi, "Inexact graph matching through graph coverage," in *Proceeding of the First International Conference on Pattern Recognition Applications and Methods*, vol. 1, Feb. 2012, pp. 269–272.

[4] G. Del Vescovo and A. Rizzi, "Automatic classification of graphs by symbolic histograms," in *Proceedings of the 2007 IEEE International Conference on Granular Computing*, ser. GRC '07. IEEE Computer Society, 2007, pp. 410–416.

[5] R. Greenlaw, H. J. Hoover, and W. L. Ruzzo, *Limits to parallel computation: P-completeness theory*. New York, NY, USA: Oxford University Press, Inc., 1995.

[6] Owens, D. John, Luebke, David, Govindaraju, Naga, Harris, Mark, Kruger, Jens, Lefohn, E. Aaron, Purcell, and J. Timothy, "A Survey of General-Purpose Computation on Graphics Hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, Mar. 2007.

[7] W. Imrich and S. Klavžar, *Product graphs, structure and recognition*, ser. Wiley-Interscience series in discrete mathematics and optimization. Wiley, 2000.

[8] S. Rajasekaran and J. Reif, *Handbook of parallel computing: models, algorithms and applications*, ser. Chapman & Hall/CRC computer and information science series. Chapman & Hall/CRC, 2008.

[9] A. Borodin, "On relating time and space to size and depth," *SIAM J. Comput.*, vol. 6, no. 4, pp. 733–744, 1977.

[10] S. Fortune and J. Wyllie, "Parallelism in random access machines," in *Proceedings of the tenth annual ACM symposium on Theory of computing*, ser. STOC '78. New York, NY, USA: ACM, 1978, pp. 114–118.

[11] L. M. Goldschlager, "A universal interconnection pattern for parallel computers," *J. ACM*, vol. 29, pp. 1073–1086, October 1982.

[12] F. Leighton, *Introduction to parallel algorithms and architectures: arrays, trees, hypercubes*. M. Kaufmann Publishers, 1992, no. v. 1.

[13] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, pp. 103–111, August 1990.

[14] P. de la Torre and C. P. Kruskal, "Submachine locality in the bulk synchronous setting (extended abstract)," in *Proceedings of the Second International Euro-Par Conference on Parallel Processing-Volume II*, ser. Euro-Par '96. London, UK: Springer-Verlag, 1996, pp. 352–358.

[15] G. Bilardi, C. Fantozzi, A. Pietracaprina, and G. Pucci, "On the effectiveness of d-bsp as a bridging model of parallel computation," in *Proceedings of the International Conference on Computational Science-Part II*, ser. ICCS '01. London, UK, UK: Springer-Verlag, 2001, pp. 579–588.

[16] L. G. Valiant, "A bridging model for multi-core computing," in *Proceedings of the 16th annual European symposium on Algorithms*, ser. ESA '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 13–28.

[17] M. Neuhaus, K. Riesen, and H. Bunke, "Fast suboptimal algorithms for the computation of graph edit distance," in *Structural, Syntactic, and Statistical Pattern Recognition. LNCS*. Springer, 2006, pp. 163–172.

[18] S. V. N. Vishwanathan, K. M. Borgwardt, R. I. Kondor, and N. N. Schraudolph, "Graph kernels," *Journal of Machine Learning Research*, vol. 11, pp. 1201–1242, Apr. 2010.

[19] M. Neuhaus and H. Bunke, "A random walk kernel derived from graph edit distance," in *Structural, Syntactic, and Statistical Pattern Recognition*, ser. Lecture Notes in Computer Science, D.-Y. Yeung, J. Kwok, A. Fred, F. Roli, and D. de Ridder, Eds. Springer Berlin / Heidelberg, 2006, vol. 4109, pp. 191–199, 10.1007/11815921_20.

[20] L. Livi, G. Del Vescovo, and A. Rizzi, "Graph recognition by seriation and frequent substructures mining," in *Proceeding of the First International Conference on Pattern Recognition Applications and Methods*, vol. 1, Feb. 2012, pp. 186–191.

[21] K. Riesen and H. Bunke, *Graph Classification and Clustering Based on Vector Space Embedding*, ser. Series in Machine Perception and Artificial Intelligence. World Scientific Pub Co Inc, 2010.

[22] I. Rao and K. Sarma, "On Tensor Product of Standard Graphs," *International Journal of Computational Cognition*, vol. 8, no. 3, p. 99, 2010.

[23] E. Sampathkumar, "On tensor product graphs," *Journal of the Australian Mathematical Society (Series A)*, vol. 20, no. 03, pp. 268–273, 1975.

[24] T. Gärtner, *Kernels for structured data*, ser. Kernels For Structured Data. World Scientific, 2008, no. v. 72.

[25] V. Volkov and J. W. Demmel, "Benchmarking gpus to tune dense linear algebra," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, ser. SC '08. Piscataway, NJ, USA: IEEE Press, 2008, pp. 31:1–31:11.

[26] Q. Hou, K. Zhou, and B. Guo, "Bsgp: bulk-synchronous gpu programming," *ACM Trans. Graph.*, vol. 27, pp. 19:1–19:12, August 2008.

[27] A. Cinti and A. Rizzi, "Neurofuzzy min-max networks implementation on FPGA," in *International Joint Conference on Computational Intalligence (IJCCI)*, ser. Neural Computation Theories and Analysis (NCTA), Paris, 24-26 November 2011.

[28] D. Kirk, W. Hwu, and W. Hwu, *Programming massively parallel processors: a hands-on approach*, ser. Morgan Kaufmann. Morgan Kaufmann Publishers, 2010.