Tree Matching on Parallel Machines using Data Shaping

Parijat Shukla*  
Iowa State University, Ames, IA 50011  
prijats@iastate.edu

Arun K. Somani  
Iowa State University, Ames, IA 50011  
arun@iastate.edu

Abstract

Real time big data analytics has become important to meet the business as well as other decision-making needs in many complex applications. A significant portion of such data is available and stored in semi-structured form. A tree-based organization is commonly used in such cases. Tree matching is a core component for many applications such as fraud detection, spam filtering, information visualization and extraction, user authentication, natural language processing, XML databases, bioinformatics, etc. Comparing ordered (unordered) trees is compute-intensive, in particular for Big Data.

To facilitate comparison of ordered trees, in this paper we address the problem of shaping the semi-structured data to enable time efficient processing on contemporary hardware such as a GPGPU (General Purpose Graphics Processing Unit) and INTEL MIC (a multi-core processor). Specifically, our data shaping approach enables pre-computation of partial edit distance values in parallel. We also develop processor-specific techniques keeping in mind compute requirements of various constituent stages of PTED. We evaluate our work using real world data sets. Our experimental results show that our SIMT-based PTED-GPU (Parallel Tree Edit Distance using GPU) implementation shows speedup of up to 12X when compared to the state-of-the-art in tree edit distance (TED) computation. In addition, our techniques when ported on CPUs are scalable. Finally, we discuss appropriateness of compute platforms w.r.t. various constituent stages of proposed PTED.

Keywords: Big Data, data analytics, tree edit distance, parallel processing, GPGPU, data shaping

1. Introduction

Data deluge or Big Data phenomenon has led to ever-increasing volumes of data of various forms such as structured, semi-structured, and unstructured. A compelling dimension of Big Data is velocity. Rapid and timely analytics on Big Data offers significant advantage to business and scientific communities. A study [Intel, 2012] predicted that in future, organizations have to deal with more and more unstructured and semi-structured data generated from sensors and devices, and that in future a majority of workload in organizations will comprise the real-time analytics.

A significant portion of such data is available and stored in semi-structured form. A tree-based organization is commonly used in such cases. Several data analytic pipelines involve tree comparison operations as their core components. For example, tree comparison between ordered (or unordered) trees, finding the edit distance between a given pair of trees (i.e. tree edit distance), etc. Finding tree edit distance in near-real-time is critical for various applications such as fraud detection [Xu et al., 2006], spam filtering [Muscat, 2011], etc. Fast tree edit distance solution is key to tasks such as information visualization [Alpuente, 2009], mining web applications [Barkol et al., 2011], automatic news extraction [Reis et al., 2004], and visual password based authentication [Okundaye et al., 2014]. Tree edit distance is also used in recognizing textual entailment between syntactic trees of texts [Kouylekov and Magnini, 2005], [Androutsopoulos and Malakasiotis, 2010], and sentence ranking [Yao et al., 2013] for the purpose of question answering.

The tree comparison problem has quadratic complexity in terms of the number of nodes in the given trees. Therefore, the structure of tree structured data presents processing time challenges. As volume of semi-structured data sets (e.g. XML, JSON, etc.) increases and real-time data analytic over such data sets become indispensable, there is a need to leverage parallel-processing platforms effectively. For example, we can accelerate the tree-matching component using parallel hardware such as a General Purpose Graphics Processing Units (GPUs) effectively.
General-purpose graphics processing units (GPUs) have become mainstay for high performance computing. Accelerators such as GPUs, not only provide thousands of parallel cores and special purpose software managed memories, they also offer significant cost and energy advantages over their multi-core CPU counterparts if the computing needs match the architecture of GPU. For example, when compared to latest quad-core CPUs, Tesla GPUs (C2050/C2070) deliver a matching supercomputing performance at 1/10th of the cost and 1/20th of the power consumption \cite{Nvidia2010}. Five out of top ten supercomputers from Top500 listing are powered by accelerators such as Intel Xeon Phi or NVIDIA K20/40 \cite{NvidiaTop5002014}. For example, Tianhe-2 (NSCC, China), listed on the top, comprises Intel Xeon Phi and Titan (ORNL, TN, USA), number two in the list, comprises NVIDIA K20 accelerators. We at ISU have a large number of nodes including both of these platforms in addition for multicore nodes. It is important to explore applicability of SIMT machines for not only to meet the high performance computing needs but also from the cost and energy efficiency point of view.

In the context of tree matching problem using GPU we observe that this problem is compute- as well as data-intensive. Finding tree edit distance between two trees has a time complexity of $O(n^3)$. Given that GPUs are rich in compute resources, they can be good fit to process tree matching problem faster. However, the compute units of GPUs must be presented data for processing in a timely and appropriate manner to gain any significant speedup. GPUs also have memory hierarchy limitations. The compute and memory resources of a GPU need to be used effectively. Moreover, substructures (e.g. subtrees) contained within the semi-structured data object are critical for parallelizing computation. These substructures can be used to perform partial computations independently (on any type of hardware). Partial computations using these substructures can be used to derive the overall outcome. Thus, it is important to identify the substructure(s) contained in semi-structured data objects and reorganize the tree-structured data appropriately.

In this paper, we address the problem of shaping the semi-structured data to enable time efficient processing on SIMT hardware. Specifically, we propose a novel data shaping technique to identify relevant substructures from a tree object. We also propose a novel approach to pre-compute the partial edit distance between trees using the substructures identified. The computations involving substructures offer a high level of parallelism and are independent of each other, and can be realized on a parallel hardware. Our data shaping technique allows a high degree of data reuse employing the available on-chip memories effectively. This reduces the traffic to off-chip memory. Our low level approach is applicable to the data structure used in the existing algorithms. We develop a general framework to speed up the data analytic pipeline in the context of semi-structured datasets. Our framework is applicable to several scenarios such as acceleration of key kernels of tree edit distance based algorithms and to calculate approximate distance between two trees, etc.

**Contributions.** This paper makes following contributions:

- A novel data shaping algorithm to identify substructures (subtrees) within a tree which enables parallel processing of trees.
- A novel approach to pre-compute the partial edit distance between trees, using the substructures identified, in parallel.
- A technique to compute tree edit distance on parallel hardware.
- A techniques to parallelise computations occurring in key stages of Parallel Tree Edit Distance (PTED).
- Analysis of compute requirements of key constituent stages of Parallel Tree Edit Distance problem and recommendation of appropriate platform for those key stages.

The rest of the paper is organized as follows: In Section 2, we discuss the background for the work and challenges involved. Next, we discuss our approach towards developing an efficient solution. Section 3 describes the tree matching process by exploiting substructure and efficient computations on GPU. Section 4 describes platform-centric optimization techniques. Section 5 and 6 present evaluation methodology adopted and the results obtained, respectively. We discuss the implications of results obtained in Section 7. Related work is discussed in Section 8. Section 9 and Section 10 provides our conclusions and future directions, respectively.

### 2. Background

We first provide an overview of the tree edit distance work and GPU and introduce some of the terminology used in the context of GPU. For more details see \cite{Nvidia2017}.

#### 2.1 Background

Tree edit distance (TED) between two trees is described in terms of the minimum work needed to transform one tree into another tree by applying a set of edit
operations. These edit operations are deletion, insertion, and substitution [Bille (2005)].

Figure 1 shows an example of tree edit distance using three trees S0, S1, and S2. Consider Trees S0 and S1. Deletion of node "v5" from tree S1 results in tree S0. Hence, tree edit distance between S0 and S1 is 1. Similarly, tree edit distance between S1 and S2 is 2. The reason being that the two operations on S1, namely (1) substitution of node “v5” to “c”, and (2) deletion of node “v4” results in S2.

Zhang and Sasha proposed a recursion-based algorithm for calculating tree edit distance between two ordered trees [Zhang and Shasha (1989)]. Their algorithm recurs on the rightmost root and operates in two stages: (1) The first stage comprises (a) naming the nodes in post order, (b) identifying key-roots, and (c) identifying leftmost descendants. (2) The second stage calculates the edit distance between the subtrees at key-roots and updates the tree edit distance matrix. At the end of this stage, edit distance between two trees is obtained. For details, refer to [Zhang and Shasha (1989)]. Number of relevant sub-problems or the time complexity involved is \( O(n^2 \times m^2) \) (which can be up to \( O(n^4) \) when \( m \sim n \) where “m” and “n” are the number of nodes in Trees F and G, respectively.

2.2 GPGPU

On a GPU, several Streaming Processors (SPs) or cores can execute in parallel within a Streaming Multiprocessor (SM). For example, in a GPUs such Tesla C2070, K-10, K-20, etc. 32 SPs (cores) execute in a lock-step manner within a SM. Cuda, OpenCL, Brook+, etc. are the programming environment to support the program development. They also include small amount of special purpose on-chip memories, called Shared Memory, Constant Memory (CM), and Texture Memory (TM) besides the regular L1/L2 cache memories.

2.3 CHALLENGES

As noted earlier, tree-matching problem is compute-intensive involving quadratic order pair-wise comparison operations. Parallel machines like GPU offer tremendous processing power. High computational complexity of the problem motivates us to use highly parallel machines like General Purpose Graphics Processing Unit (GPU) for such compute intensive problems.

A key observation here is that the data items are required to be present close to the cores (streaming processors) working on those data items. If this can be achieved then additional computations can be launched for almost free, leading to increase in the data reuse, which results in a higher Compute to Global Memory Access (CGMA). The required memory bandwidth and system wide data movement are further constrained by energy consumption and power dissipation limits. Reduced system wide data movement allows increased power allocation to computation cycles. However, increasing data reuse factor necessitates reconsideration of the algorithm to be used.

GPUs posses several architectural characteristics, which make tree processing challenging: (1) Lock-step execution of the streaming processors (SPs) in streaming multiprocessor (SM). (2) High memory access latency associated with device memory (or main memory on a GPU). (3) Limitations of memory hierarchy of the GPUs,
for e.g., data access constraints associated with constant memory.

To work with these architectural features effectively several issues need to be addressed that include:
(1) Size of the computation executed in parallel,
(2) Size of the data item to be processed in parallel, and,
(3) Pattern and order of accessing data.

3. Tree Matching on Parallel Hardware

Our solution to the tree matching problem operates in three stages:
Stage 1: Subtree Identification. This stage results into a collection of subtrees. This is a data shaping stage.
Stage 2: Computation of partial distances. Compute partial distance values using Subtrees. Also, compute distance between leaf nodes. Record these pre-computed values in TreeDistance Matrix. We use parallel processing to speed up the computation process.
Stage 3: Computation of final Tree Edit Distance. Compute final tree edit distance using the pre-computed edit distance values (from Stage 2).

We describe these stages in detail in the following.

Algorithm 1: Identify Subtree

Left[k]: Leftmost leaf descendant of the subtree rooted node at k.
Keyroot[ ]: Array of keyroot nodes.
N: Node currently being visited.
1: Visit Keyroot nodes one by one
2: for every N do
3: if \((\text{depth}[N] - \text{depth}[\text{left}(N)]) == 1\) then
4: Include nodes left[N]...N as a subtree
5: end if
6: Mark N as a visited node
7: end for

3.1 Subtree Identification

The basic idea for subtree identification is to organize tree structured data in order to facilitate efficient computations on the GPU and to create opportunities for parallelism and exploiting memory hierarchy. After shaping tree data into subtrees (substructures) we can use these substructures for matching a pair of trees. Data shaping also helps exploit the compute and memory resources of the underlying GPU hardware effectively. Subtree Identification Stage is executed on CPU.

Subtree Identification Algorithm. We decompose a given tree into subtrees using a subtree encoding algorithm as shown in Algorithm 1. Figure 2(a) depicts Tree T1. Definitions of “Left[]” and “Keyroot[]” are similar to prior works \cite{Zhang:1989:DTS, Demaine:2007:NSP}, etc. Left[i] is the post order traversal number of leftmost leaf descendant of the subtree rooted at index “i”. A Keyroot is a node of a given Tree, T such that either has a left sibling or is the root of T. Some nodes in T1 are enclosed in dotted ellipse. These are set of nodes identified by Algorithm 1 as subtrees, namely T1-1, T1-2, and T1-3. Figure 2(b) shows Tree T2 and its three subtrees T2-1, T2-2, and T2-3. Next, we explain the subtree encoding process.

Figure 2: (a) Tree T1 and its three subtrees. (b) Tree T2 and its three subtrees. Nodes are recorded in post-order traversal number which are shown in gray circles.

Illustration. Consider a post-order traversal of Tree T1 as shown in Figure 2(a). Apply Subtree Encoding Algorithm to T1. We obtain three subtrees. Nodes (c, d, b) constitute first subtree T1-1. Nodes (h, i, g) are part of second subtree, i.e. T1-2. Nodes (l, m, n, k) form the third subtree T1-3. For Tree T2 (shown in Figure 2(b)) subtree identified by tree encoding algorithm are as follows: We obtain three subtrees. Nodes (c, d, b) constitute first subtree T2-1. Nodes (f, e) are part of subtree T2-2, and nodes (l, m, n, k) constitute subtree T2-3.

Memory requirement. Next, we analyze the space complexity of data structures used in Identify Subtree Algorithm. The Subtree Encoding Algorithm stores every node of Tree T in NodeList exactly once. Hence, space complexity of NodeList is \(O(n)\). Total size of metadata associated with subtrees (such as start and end indices of subtrees) depends upon the number of subtrees identified. Note that total number of subtrees possible is \(\leq O(n)\). Hence, the overall space complexity of Subtree Encoding Algorithm is \(O(n)\).
3.2 Computing Partial Tree Edit Distance Values Using Subtrees

We compare every subtree of T1 with every subtree of T2 and compute the edit distance between them. All of these subtree comparison operations are independent of each other and can be realized in parallel. We use parallel computational resources of SIMT machine to compute the edit distance. We record the pre-computed values of subtree edit distances in matrix M[][]]. Subtree edit distance is basically the ordered string edit distance, namely Levenshtein Distance [Levenshtein 1966]. The cost matrix C[][]] is also calculated by the GPU. Listing 1 depicts the GPU Kernel used to realize this computation. Block “i” obtains the start index of “i”th subtree from a redirection array sub1a[] (see Line 5). Similarly, Thread “j” obtains the start index of “j”th subtree from a redirection array sub2a[] (see Line 8). Start indices of subtrees of T1 and T2 are maintained in array sub1[] and sub2[], respectively. Depending upon the nature of subtrees (i.e. single node/multi node), the Threads update the matrix M. We describe this process shortly.

Listing 1: Compute Partial Distance on GPU

```c
__global__ void ComputePartialDistGPU( . . . )
{
    // variable declarations ... 
    if(blockIdx.x < m)
        if(sub1[sub1a[blockIdx.x+1]]>0)
            i = sub1a[blockIdx.x+1];
        if(threadIdx.x < n)
            if(sub2[sub2a[threadIdx.x+1]]>0)
                j = sub2a[threadIdx.x+1];
        if(i==sub1[i]) //t1 is a single node subtree
            //Find distance between t1 and t2, Update M.
        else //t1 is a multi node subtree
            if(j==sub2[j]) //t2 is a single node subtree
                //Find distance between t1 and t2, Update M.
            else //t2 is also a multi node subtree
                if((left1[i-1]==left1[sub1[i-1]])
                    &&(left2[j-1]==left2[sub2[j-1]]))
                    //Case 2: Both are tree
                    M[i+SZ]+j = dist2(word1, len1, word2, len2);
                else
                    if((left1[i-1]==left1[sub1[i-1]])
                        &&(left2[j-1]==left2[sub2[j-1]]))
                        //Case 3: One is not a tree
                        M[i+SZ]+j = dist3(word1, len1, word2, len2,
                            sub1[i-1], sub2[j-1], SZ, M);
                
                //End of Thread
            
        //End of Block
    
    return;
} //End of kernel
```

Edit Distance Matrix. Figure 3 depicts the edit distance matrix, M[][]] mentioned in Listing 1. M is a 14×10 matrix. The number of rows and columns depends on the number of nodes in T1 and T2, i.e., T1 has 14 nodes and T2 has 10 nodes, therefore dimension M is 14×10. First row and first column marked in green are indices. Note that some cells in M[][]] are marked as gray. These cells correspond to the entries which are calculated using the subtrees identified in Stage 1 (refer Section 3.1). For example, the values in cells M[1:3][1:3] correspond to subtrees T1-1 and T2-1. This set of cells are indicated by a top-left box marked by dotted lines. Similarly, values in cells M[1:3][4:5] correspond to T1-1 and T2-2 marked by top-middle box, and values in cells M[1:3][6:9] marked by top-right box correspond to T1-1 and T2-3.

Figure 3: Pre-computation of edit distance values using subtrees.

Computation Assignment on to GPU. Figure 3 also depicts the assignment of computations using subtrees to various Blocks and Threads of GPU. The computation of edit distance values between T1-1 and T2-1 is assigned to Thread0 of Block B0. Computation for T1-1 and T2-2 pair is assigned to Thread1 of Block B0, and computation of T1-1 and T2-3 pair is assigned to Thread2 of Block B0. Similarly, Blocks B2 and B4 are assigned computations involving subtrees T1-2 and T1-3, respectively. In Figure 3 also note that Block B1 is assigned edit distance computation between node “f” (of T1) and subtrees of T2. Similarly, Block B3 is assigned the distance computation between node “j” (of T1) and subtrees of T2.

Edit Distance Matrix Computations. We explain the calculation of edit distance matrix, M[][]] using three subtrees of T1 and three subtrees of T2 for comparison. Note that subtree edit distance between pairs T1-1 and T2-1, T1-2 and T2-2, T1-3 and T2-3 is zero. The reason being that the subtrees in these pairs are identical. Refer to Figure 2(a)-(b). Edit distance between T1-2 and T2-2 is obtained by replacing first two nodes of T1-2 i.e.
nodes (a, b) by nodes (e, f) of T2-2 and then deleting the remaining nodes of T1-2 i.e. nodes (c, d). Other entries of matrix M can be obtained in the similar way. Figure 4 depicts the subtree edit distance computation process for T1-1 and three subtrees of T2 in Figure 4(a)-(c). Figure 5 depicts the partially computed M\[\].

Figure 4: Computation of subtree edit distance values. (a) shows computation of edit distance values between t1-1 and t2-1. (b) and (c) shows the computation for t1-1 and t2-2 pair, and t1-1 and t2-3 pair, respectively.

3.3 Final Tree Edit Distance

The final tree edit distance calculated using dynamic programming model similar to the one mentioned in earlier works Zhang and Shasha [1989], Demaine et al. [2007], Pawlik and Augsten [2011]. The routines TED() (Listing 2) and Compute_Fdist() of Listing 3 depicts the dynamic programming formulation. We differ from previous studies (namely Zhang and Shasha [1989], Demaine et al. [2007], Pawlik and Augsten [2011]) in one key aspect – we do not have to compute all the values, since the matrix M is partially filled already. This is due to the precomputed values available from previous stage (refer Section 3.2). Line 8 of routine TED() (Listing 2) supports this by an “if” condition. For computing distance between two forests, we use the method proposed in prior works Zhang and Shasha [1989], and is shown as Listing 3.

Listing 2: Computation of Final Distance

```
1 TED() {
2     int i1, j1; int i, j;
3     for(i1=1;i1<total_key_roots1;i1++)
4         for(j1=1;j1<total_key_roots2;j1++)
5             i = key_roots1[i1]; j = key_roots2[j1];
6             //Check if Treedist value for this
7             //pair of key roots already exist.
8             if (M[i][j] > 0) continue;
9         else {Compute_Fdist(i,j,left1,left2,C);}  
10     }
```

Listing 3: Computation of Forest Distance (Zhang and Shasha [1989])

```
1 Compute_Fdist(int pos1, int pos2,
2      int left1[], int left2[], int C[][]) {b1=pos1-left1[pos1]+2; b2=pos2-left2[pos2]+2;  
3     int dist[b1][b2]; int i, j, k, m, n;
4     for(i=0;i<b1;i++) for(j=0;j<b2;j++) {dist[i][j]=0;}
5     dist[0][0]=0;
6     for(i=1;i<b1;i++) dist[i][0]=dist[i-1][0]+C[i][0];
7     for(i=1;i<b2;i++) dist[0][i]=dist[0][i-1]+C[0][i];
8     for(k=left1[pos1], i=1; k<=pos1; k++, i++)
9         for(l=left2[pos2], j=1; l<=pos2; l++, j++)
10             if both are trees, then:
11                 if ((left1[k]==left1[pos1])
12                     &&(left2[l]==left2[pos2]))
13                     dist[i][j]=MIN(dist[i-1][j]+C[0][l],
14                                    dist[i][j-1]+C[k][0],
15                                    M[k][l] + dist[i][j]);
16             else {m = left1[k] - left1[pos1];
17                 n = left2[l] - left2[pos2];
18                 dist[i][j]=MIN(dist[i-1][j]+C[0][l],
19                                dist[i][j-1]+C[k][0],
20                                dist[m][n]+M[k][l]);}
21         return;
22 }
```

4. Platform-centric Optimization

State-of-the-art computing landscape exhibit a significant level of heterogeneity. There are GPUs, which comprise thousands of lightweight cores, and MIC architecture-based accelerators with 60-72 cores. It is also likely that there would be throughput oriented cores (TOCs) and latency optimized cores (LOCs) on the same die. Today’s CPUs comprise multiple cores, up to 32 cores and beyond, on the same chip. There are System on Chips (SoCs) architectures which comprise several cores, up to 4-8, hundreds of graphic cores which run in parallel, and various type of fast memories, all on the same die.
In general, data movement between CPU and accelerators (GPU, MICs) is costly, both in terms of latency and energy. In some scenarios, the data movement from CPU to GPU does not make sense and must be better avoided. This is true especially when compute requirements of our parallel tree edit distance technique are on the lower side and which could be addressed adequately by compute resources (cores) available on the CPU side.

We define application portability or performance portability as follows: application should be portable across a set of platforms and the addition or removal of compute resources (such as cores/threads) on those platforms should result in an increase or decrease of performance. Performance portability is possible as long as target platform meets basic performance critical criteria for the application. One such key criteria is parallelizability.

Focus of this section is to understand the performance parallelizability aspect of Parallel Tree Edit Distance (PTED) computation technique w.r.t. to various modes of parallelizability available on state-of-the-art computing platforms. We are especially interested in knowing how Stages 2-3 can be benefitted by parallelism available on CPUs, manycore architectures, such as KNLs/KNHs, and others.

Specifically, we explore performance parallelizability aspect of our parallel tree edit distance technique. In this context, we consider three techniques: (1) Parallelization of Partial Edit Distance Computation on SIMT architectures (i.e., GPUs), and (3) Shared memory-based optimization.

4.1 Parallelization of Partial Tree Edit Distance on CPU

Computation of edit distance values between subtrees of T1 and T2 can be done concurrently. This aspect has been leveraged in Listing 3 where, for example, computations of edit distance values between T1-1 and T2-1 is assigned to Thread0 of Block B0. Similarly, the computation for T1-1 and T2-2 pair is assigned to Thread1 of Block B0, and computation of T1-1 and T2-3 pair is assigned to Thread2 of Block B0, and so on. Refer to Section 3.2 and Figure 3.

OpenMP or Open Multi-Processing is an API that supports shared memory programming [Chapman et al., 2008]. OpenMP relies on compiler directives, library routines, and environment variables to manage run-time behavior. OpenMP is runtime of choice for programming multicore CPUs, manycore architectures, such as KNLs/KNHs, and others.

Listing 4: Compute Partial Distance on Manycore

```
ComputePartialDistManyCore(...) {
    int i, j, k; int single_match;
    int len1, len2;
    int i1, j1;
    int word1[100]; int word2[100];
   // for omp:
    int tid, nthreads;
    int chunk = 0; int S = 1024;
    int i_idx; int j_idx;
    int blockIdx_x = 0; int threadIdx_x = 0;
    omp_set_dynamic(0);
   #pragma omp parallel shared(nthreads)
    private(i, j, k, single_match, len1, len2, word1, word2, i1, j1)
    num_threads(4)
    (tid = omp_get_thread_num());
    if ( tid == 0 )
        nthreads = omp_get_num_threads();
    
   #pragma omp for schedule(static) 
    for ( blockIdx_x = 1; blockIdx_x <= ...
            doca_num_nodes; blockIdx_x++)
    {
        i = subtree1[subtree1a[blockIdx_x] > 0]
        j = subtree2[blockIdx_x ];
        if ( threadIdx_x = 1; threadIdx_x <
            doc2a_num_nodes; threadIdx_x++)
        {
            if(sub2[sub2a[threadIdx_x+1]] > 0)
            |j= sub2a[threadIdx_x+1];
            if(i == sub1[i]) // t1 is a single node subtree
            (i == sub1[1]) ||/t1 is a single node subtree
                Find distance between T1 and T2, Update M.
            else // t1 is a monitode subtree
                (i == sub1[1]) ||/t2 is a single node subtree
                Find distance between T1 and T2, Update M.
            else // t2 is also a monitode subtree
                (i == sub1[1]) || (i == sub1[1])
                (i = left2[j-1] = left2[sub2[j-1]]))
                // Case 2: Both are tree
                M[i+SZ+j] = dist2(word1, len1, word2, len2, j);
            else
                (i == left1[i-1] = left1[sub1[i-1]])
                &&(left2[j-1] = left2[sub2[j-1]]))
                // Case 3: One is not a tree
                M[i+SZ+j] = dist3(word1, len1, word2, len2, sub1[i-1], sub2[j-1],SZ,M);
        }
    } } 
} } 
```

In order to exploit large number of cores available on multicore/manycore, accelerators, the set of independent computations are assigned to cores. This assignment is realized via OpenMP runtime. Listing 3 depicts the OpenMP version to compute partial edit distance values.
From Listing 4, we observe that the computations are organized as a two level nested loop. The "pragma openmp static" on Line 26 is a directive for OpenMP compiler to parallelize the loop body following it. Typically, the loops will be chunked into groups. The chunk size depends on number of loop iterations and number of OpenMp threads provisioned. Typically, one OpenMP thread is assigned to a single core, unless hyperthreading or symmetric multithreading is used.

A key criteria for loop parallelization is that the number of loop iterations should be countable at compile time. The data shaping stage ensures this. The total number of loop iterations depend on number of sub-trees identified from the input trees.

Consider the computation illustrated in Figure 5. There are total fifteen loop iterations, five in outer loop and three in inner loop. When using OpenMP parallel runtime, the iterations in outer loop are chunked and assigned to OpenMP thread where each OpenMP thread, typically, runs on a single core of the 16/32 core CPU.

4.2 Parallelization of Final Edit Distance Computation

Recall that in sequential version (Listing 3) the computation proceeds sequentially from left to right, and top to bottom. This sequential execution is assigned a single thread (or a CPU core). The Final edit distance computation can be parallelized by identifying set of independent computations which can be executed concurrently. To this end, we make two key observations:

Observation 1. First observation is as follows: There exist a dependency between the values in cells in same row (and same column). Due to this the row-wise (and column-wise) computations cannot be parallelized. For example, in order to compute value for cell M[i][j-1], M[i-1][j], and M[i-1][j-1] are required.

Observation 2. Second key observation is that the values in cells along diagonal, (refer to double-edged arrows in Figure 6), do not exhibit any dependency. Alternatively, there is no dependency between intra-diagonal values. These intra-diagonal values can be computed in parallel. There exist dependencies between the inter-diagonal values. The computation proceeds as waves of diagonals and we refer to this parallelization approach as wavefront-based approach. Next, we describe the parallelization technique.

Figure 6 describes the parallelization based on wavefront approach. The figure depicts sixteen iterations denoted as Iter1, Iter2, · · · , Iter16. The figure also details how intra-iteration computations are assigned to individual threads/cores. Consider the first iteration denoted by Iter1. In Iter1, there is only computation, i.e., value for only single cell is to be computation. And this computation is assigned to a single thread, Thd1. In second iteration, i.e., in Iter2, values for two cells are to be computed and it is handled by two threads, namely, Thd1 and Thd2. Similarly, in tenth iteration, values for ten cells are to be computed and ten threads are needed. Ten threads, Thd1, Thd2, · · · ,Thd10, are assigned one computation each.

Maximum number of threads needed at any point do not exceed MAX(b1, b2). Total number of iterations depend on the size of subtrees involved in forest distance computation. Specifically, total number of iterations = b1 + b2 -1, where b1 and b2 are the range of nodes in trees T1 and T2, respectively. To facilitate (intra-diagonal) parallel computations, indices of data are calculated based on thread index, iteration number, b1, b2, etc. The indices calculation is an overhead.

Listing 5 depicts implementation of wavefront-based parallelization of forest distance computation. Lines 1-18 of Listing 5 is similar to the sequential version depicted in Listing 3. Only notable difference is in Line 5 which includes omp pragma for parallel region.

Line 19 of Listing 5 initializes the following variables to 0: index, kk, initK, initL, max, mid_part, mid_width, num_steps_top, num_steps_mid, num_steps_bot, num_steps, type1, type2, max, mid_part, mid_width, num_steps_top, num_steps_mid, num_steps_bot, num_steps, type1, type2.

Line 20-28 determine how to calculate parameters involved in parallelization.
Listing 5: Compute_Fdist_Parallel

```c
int Compute_Fdist_Parallel()
{
    // i_prime, j_prime, i_out, j_out;
    int kl = 1; int k2 = 1;
    float compute_intensity=0.0; int b1, b2;
    #pragma omp parallel private(i_p,j_p,i_out,j_out)
    for(i_p=1; i_p<=total_key_roots1; i_p++)
    {
        if(j_p=1; j_p<=total_key_roots2; j_p++)
        {
            //pragma omp single
            i_out=key_roots1[i_p]; j_out=key_roots2[j_p];
            if(tdist1[key_roots1[i_p]] > 0) { debug_count1 ++ ; }
        }
    } // parallel version (for manycore/omp)
    for(i_p=1; i_p<=num_steps_top; i_p++)
    {
        num_steps_top = b2; num_steps_mid=b1;
        if(tdist1[key_roots1[i_p]] > 0) { debug_count1 ++ ; }
    }
    for(i_p=1; i_p<=num_steps_mid; i_p++)
    {
        num_steps_mid = b2;
    }
    for(i_p=1; i_p<=num_steps_bot; i_p++)
    {
        num_steps_bot = b1;
    }
}
```

Note that the forest distance computation is realized in three parts: first part, mid part, and last part. Line 34-43 deal with the first part. Line 36 includes a directive to parallelize computations to CPU cores.

Line 44-45 outlines the sketch for mid part. Parameters involved in mid part are calculated as following. For type1: i=(b2+num_steps_bot+loop_idx) and j=initK+i; l=initL+j-1. For type2: i=b1+1-loop_idx; j=(num_steps)+loop_idx; k=initK+i; l=initL+j-1. Parameters involved in last part are calculated as following. For type1: i=(b1+1)-loop_idx; j=(1+num_steps_bot+loop_idx)+loop_idx; k=initK+i; l=initL+j-1. For type2: i=(b1+1)-loop_idx; j=b2+1+num_steps_bot+loop_idx; k=initK+i; l=initL+j-1.

Listing 6: Helper routine

```c
Subroutine()
{
    if ((left1[k]==left1[pos1])&&left2[i]==left2[pos2])
    {
        sh_f_dist_2d[i][j] = MIN(sh_f_dist_2d[i][j-1]+cost_matrix[0][1], sh_f_dist_2d[i][j-1]+cost_matrix[k][0], sh_f_dist_2d[i-1][j-1]+cost_matrix[k][1]);
    }
    else
    {
        m = left1[k] - left1[pos1];
        n = left2[i] - left2[pos2];
        sh_f_dist_2d[i][j] = MIN(sh_f_dist_2d[i-1][j]+cost_matrix[0][1], sh_f_dist_2d[i][j-1]+cost_matrix[k][0], sh_f_dist_2d[i-1][j-1]+cost_matrix[k][1]);
    }
}
```

4.3 Shared Memory-based Optimization

GPUs, along with regular cache hierarchy and constant memory, also provide on-chip shared memory. For example, in Kepler K20 GPU, a total of 64 KB of fast memory is provided on every SM. For details see [Nvidia 2013]. Common use cases of shared memory provided on GPUs is to hide irregular memory accesses, and for memory-based block level synchronization across.

The shared memory on GPUs can be leveraged to accelerate forest edit distance computations in the following way: provision a shared memory data structure in GPU Kernel which holds forest distance matrix (refer fdist[[]] in Listing 3). By provisioning fdist[[]] as a shared memory data structure, access to global memory are reduced and while also not incurring any unwarranted memory access issues due to access patterns (reason for which is that shared memory supports irregular memory access patterns. Using shared memory in the manner
discussed above yields gain in performance. However, one drawback of this approach being the limited size of
shared memory.

A 64 KB of shared memory is good enough to store integer Fdist[][] matrix of size 128 x 128. This limitation
could be addressed by making use of shared memories
available on other SMs. This is possible by distributing
the forest distance computation on to the SMs available
on GPU. However, this can be problematic as inter block
synchronization is done via global memory as L2 caches
are not big enough to hold the matrix. Another limitation
is that blocks are scheduled arbitrarily on GPUs. To
ensure any any ordering between the blocks execution
we have a rely on device level synchronization mecha-
nisms which could easily nullify any performance gain
derived by using shared memory. We do not pursue this
line of optimization.

5. EXPERIMENTAL SETUP

5.1 DATA SETS AND PLATFORM

We use three popular xml data sets in our experiments
namely Nasa, SwissProt, and tree_bank_e [15].
“Nasa” is an astronomical dataset having average depth
and maximum depth of trees as 5.5 and 8, respectively.
“Swissprot”, a curated protein sequence database, has
average depth and the maximum depth of trees as 3.5
and 5, respectively. Tree_bank_e, also referred in this
paper as “Treebank”, is a dataset of English sentences.
The “Treebank” dataset is tagged with parts of speech
and is partially encrypted, with the average depth and
the maximum depth of trees in dataset as 7.8 and 36,
respectively.

We select tree pairs at rational size intervals and mea-
sure the cycles consumed and the time elapsed on GPU.
For a given tree size n we select the two trees in the
dataset that are closest to n. The value of size used in
the graphs is the average size of the two trees.

5.2 COMPUTING PLATFORMS

For experiments we used a CPU/GPU based platform.
CPU is a Linux Machine running Intel(R) Xeon(R) CPU
X5650 @2.67GHz with 24 GB RAM. The GPU used is
da Tesla C2070 device having 448 Streaming Processors
(SPs). OpenMP version 4.0 is used. We used Kepler
K20 GPUs for some of the experiments. The K20 GPU
comprises 2496 Cuda cores @ 746 MHz. Stage 1 runs on
CPU. Stage 2 and Stage 3 runs on CPU/GPU.

We compare our work to RTED. We obtained a copy
of RTED implementation from [Pawlik and Augsten][11]
and run on CPU. The RTED implementation requires the
trees to be represented in bracket notation. For example,

6. RESULT

In this section, we discuss the results of the exper-
iments for Swissprot, Treebank, and Nasa datasets.
Specifically, we discuss the performance of different
stages of PTED-GPU and speedup of PTED-GPU
implementation to RTED. We also report the performance
of partial edit distance and parallel forest distance compu-
tation algorithms in on multicore CPU.

6.1 PERFORMANCE OF PTED-GPU

Total execution time is sum of three components
namely, (1) XML parsing and Data Shaping time i.e.
Stage A (referred as Data Shaping Time), (2) Overhead
associated with GPU specific system calls (referred as
GPU overhead) (3) partial edit distance computation on
GPU (Stage B) and Final Tree Edit Distance computation
(Stage C). We do not consider GPU specific overhead
in our performance analysis since it is not due to our
PTED-GPU solution. Moreover, GPU specific overhead
can be ignored when several tree matching operations
are pipelined.

We calculate actual execution time as: actual execu-
tion time = total execution time – GPU specific overhead.
We use actual execution time (often referred as execution
time) in analysis. Figure 7 depicts the breakdown of time
spent in various stages for “nasa” dataset. From Figure 7
we observe that XML parsing and data shaping take 6
milliseconds on an average. Overhead associated with
GPU specific system calls (referred as GPU overhead)
is around ~218 milliseconds. Finally, actual execution
time is 25, 22, and 25 milliseconds, respectively, for tree
pairs having 100, 150, and 200 as the average number of
nodes. The execution time elapsed for tree pairs with
300 nodes as average number of nodes is around 35
milliseconds. For tree pairs with 400 average nodes, the
execution time is around 55 milliseconds.

Figure 8 shows the breakdown of the time spent in
various stages of PTED-GPU for “Swissprot” dataset.
From Figure 8 we observe that XML parsing and data
shaping take 7 milliseconds on average for tree pairs
having 100 to 800 as the average number of nodes. The
time elapsed in parsing and data shaping for tree pairs having 1000 and 1100 as the average number of nodes is \(\sim 18\) milliseconds. Overhead associated with GPU specific system calls is \(\sim 218\) milliseconds.

**Figure 9** shows the breakdown of the time spent in various stages of PTED-GPU for “Treebank” dataset. From **Figure 9** we observe that XML parsing and data shaping take 6 milliseconds on average. Overhead associated with GPU specific system calls is \(\sim 219\) milliseconds.

### 6.2 Scalability of PTED-GPU

**Figure 13** plots the comparison of rise in execution time vs. rise in average number of nodes for “nasa”, “Swissprot”, and “Treebank” datasets. From this Figure we note that:

1. For “nasa” dataset, increasing average number of nodes in tree pairs by 4X results in 2.3X increase in execution time. This indicates a sub-linear relationship between increases in execution time vs. increase in average number of nodes in tree pairs.

2. For “Swissprot” dataset, increasing average number of nodes in tree pairs by 11X results in 9.25X increase in execution time indicating a sub-linear relationship between the execution time vs. average number of nodes in tree pairs.

3. For “Treebank” dataset, increasing average number of nodes in tree pairs by 3X results in 1.2X increase in processing time, indicating a sub-linear relationship between the execution time vs. average number of nodes in tree pairs.

These observations reveal that our PTED-GPU implementation is scalable for a wide set of problem sizes.

### 6.3 Speedup: PTED-GPU vs. RTED

Our experiments show that, in general, PTED-GPU achieves a speedup of up to 12X over the RTED. Note that we do not consider the GPU specific overhead while comparing performance of PTED-GPU to PTED. **Figure 10**, **Figure 11**, and **Figure 12** depicts the performance improvement of PTED-GPU implementation over the RTED (ignoring GPU specific overhead) for “Nasa”, “Swissprot”, and “Treebank” datasets, respectively. From these figures we observe that:

1. For “nasa” dataset, the speedup obtained varies from 7X to 12X, approximately.

2. For “Swissprot” dataset, the speedup obtained varies from 6X to 11X (approx.). We also observe that for tree pairs having average number of nodes in 100-500 range, the speedup in performance is 10X, approximately.

3. For “Treebank” dataset, the speedup obtained is up to 11X (approx.). We also observe that for tree pairs having average number of nodes in 200-300 range, the speedup in performance is more than 10X, approximately.

### 6.4 Performance of Partial Edit Distance on CPU

We use one OpenMP thread per CPU core. We do not use hyper-threading. We set the AFFINITY = COMPACT to make sure that consecutive OpenMP threads bind to consecutive cores on the CPU.

**Figure 14** depicts the performance of partial edit distance computations on CPU cores. X-axis represents a number of OpenMP threads. Y-axis represents time elapsed in the execution of 1000 iterations of a partial edit distance computations for a 380x418 node trees from a SwissProt dataset. We report average of time recorded in ten execution.

From **Figure 14** we observe that increase in the number of OpenMP threads does reduce the time elapsed in execution. For example, by increasing number of OpenMP threads from 1 to 4, the time elapsed decreases by 3.3X. In other words, resulting speedup is \(\sim 3.3X\) as the number of cores are increased from one to four. By increasing number of OpenMP threads (CPU cores) from four to eight, we observe a decrease in time elapsed by...
Figure 10: Speedup of PTED-GPU over RTED for “nasa” dataset.

Figure 11: Speedup of PTED-GPU over RTED for “Swissprot”.

Figure 12: Speedup of PTED-GPU over RTED for “Treebank”.

Figure 13: Processing time (Data Shaping time + Execution time) of PTED-GPU for nasa, SwissProt, and Treebank.

~1.5X.

Alternatively, a two-fold increase in the number of cores yields a speedup of ~1.5X. Further increase in the number of OpenMP threads, i.e. from eight threads to sixteen threads result in a speedup of 1.11X. Finally, an increase from sixteen threads to thirty-two threads result in 1.05X, almost no speedup. The performance results indicate that partial edit distance computation shows good speedup for up to eight cores – 8X increase in compute resources yields a speedup of ~5.1X.

Use of sixteen- or thirty-two-cores should be avoided. Adding cores beyond eight cores do not yield speed up due to three reasons:

1. The partial edit distance computation is not compute intensive. Manipulation operations comprise mostly additions/subtraction, comparison.

2. Workload assigned to OpenMP threads is unbalanced. Some threads have to more work in comparison to other threads. The reason being that size of all the subtrees is not same, thus resulting in an unequal amount of work. Subtree sizes depend on the structure of the trees. This is something which is beyond the scope of data shaping algorithm.

(3) Overhead due to OpenMP runtime incurred in creation, launch, and teardown of additional threads negates any speedup gained due to adding more threads/cores. (Note that there is a finite amount of overhead involved in creating parallel regions, task distribution, and synchronization at the end of a parallel region. Refer to OpenMP literature for more details.).

Figure 14: Scalability of partial edit distance computation on cpu-time elapsed on partial edit computation for a 380×418 node tree. we report time incurred in 1000 iterations.

The thirty-two core node is actually a two socket node with sixteen cores per socket. In this setup, when thirty-two OpenMP threads are invoked, synchronization and data movement across the socket also shows its effect, thus limiting the speedup.

6.5 PERFORMANCE OF PARALLEL FOREST DISTANCE COMPUTATION ON CPU

In these experiments as well we used one OpenMP thread per CPU core, do not use hyper-threading, and set the AFFINITY = COMPACT (to ensure that consecutive OpenMP threads bind to consecutive cores on the CPU).

Figure 15 depicts the final edit distance computation
stage when forest edit distance computations are executed in parallel on CPU. X-axis represents an average number of nodes in trees involved. Y-axis represents time elapsed in milliseconds. The figure plots the time elapsed in the computation of final edit distance, which is denoted by “Total time”. The time elapsed in final edit distance is sum total of time incurred in the computation of all constituent forest edit distances. A description of trees used in this set of experiments is provided in Table 1.

From Figure 15, we observe that execution time increases with increase in the number of nodes involved in forest edit distance computation. As the average number of nodes in the input trees is increased from 100 to 1100, total time elapsed in the computation of final edit distance increases from 1.7 milliseconds to 93.9 milliseconds. Overall, for an 11X increase in an average number of input trees, there is an ∼55X increase in execution time.

Figure 15 also depicts the time elapsed in the computation of largest forest denoted by “Time_largest_forest”. The largest forest computation involves all the nodes in both trees and represents the maximum degree of parallelism generated via our parallel version of forest distance algorithm.

Figure 15 also reveals that time elapsed in the computation of largest forest is a significant fraction of total time elapsed in final edit distance computation. Specifically, for trees with an average number of nodes 100, the time incurred in largest forest distance computation is ∼0.6 millisecond. For trees with an average number of nodes as 150, 200, and 300, time elapsed in largest forest distance computation are 1.2, 1.9, and 6.6 milliseconds, respectively. For three combination of trees with 400 as average nodes, time spent in the computation of largest forest is 4.7, 9.0, and 8.8; note the difference in time elapsed (will discuss that shortly). Overall, the time elapsed in forest distance computation accounts for nearly 35% to 66% of the total forest distance computation time. Also, if input trees are similar in size the time elapsed in largest forest distance computation is nearly one-third of the total time spent in final edit distance stage.

Figure 15 also reveals that the nature of input trees affects the forest distance computation time. Specifically, the equivalence (or otherwise) in number of nodes of input trees relate to the time elapsed in the computation of largest forest and total time in final edit distance alike. As an example, consider the size of input trees listed in entries 5, 6, and 7 of Table 1. Note that an average number of nodes in these three entries is ∼400. However, for given trees, the total time and time incurred in the computation of largest forest distance is not comparable. The total time for entry #5 with input sizes (192, 591) is ∼9.8 milliseconds whereas the total time for entry #6 with input sizes (380, 418) is 25.3 milliseconds, which is ∼2.5X higher. Similarly, the time incurred in the computation of largest forest distance for entry #5 and entry #7 are ∼4.7 milliseconds and ∼9 milliseconds, respectively, which is a difference of ∼1.9X.

In practice, input trees may or may not be similar in terms in number of nodes. Therefore it is important to understand the effect of variation in input, more specifically the skewness in input sizes, on the performance of final edit distance computation.

Figure 16 captures the effect of skewness in input in terms of difference in number of tree nodes. X-axis denotes the degree of skewness. The degree of skewness is calculated as follows: Skew = ABS(DIFF(t1, t2)) / MIN(t1, t2).

Skewness is the ratio of absolute value (ABS) value of the difference (DIFF) in input sizes, and minimum (MIN) of input sizes. The input sizes are number of nodes in input trees denoted as t1 and t2. A skewness of 0.0 indicates that number of nodes in input trees are equal. And skewness of 1.0 indicates that number of nodes

<table>
<thead>
<tr>
<th>ID</th>
<th>nodes in T1</th>
<th>nodes in T2</th>
<th>Avg.</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>117</td>
<td>119</td>
<td>118</td>
<td>Nasa</td>
</tr>
<tr>
<td>2</td>
<td>192</td>
<td>203</td>
<td>197</td>
<td>Nasa</td>
</tr>
<tr>
<td>3</td>
<td>380</td>
<td>418</td>
<td>399</td>
<td>Nasa</td>
</tr>
<tr>
<td>4</td>
<td>311</td>
<td>514</td>
<td>412</td>
<td>Nasa</td>
</tr>
<tr>
<td>5</td>
<td>192</td>
<td>591</td>
<td>391</td>
<td>Nasa</td>
</tr>
<tr>
<td>6</td>
<td>380</td>
<td>418</td>
<td>399</td>
<td>Nasa</td>
</tr>
<tr>
<td>7</td>
<td>311</td>
<td>514</td>
<td>412</td>
<td>Nasa</td>
</tr>
<tr>
<td>8</td>
<td>1130</td>
<td>1076</td>
<td>1103</td>
<td>Swissprot</td>
</tr>
</tbody>
</table>

Figure 15: Scalability of forest distance computation on cpu.

Table 1: Characteristics of trees used in parallel forest distance experiment (Section 6.5)
Figure 16: Skewness in input sizes vs. fraction of Time elapsed in computations of largest forest.

differ by 1X. Y-axis denotes the ratio of time elapsed in the computation of largest forest vs. total time elapsed in final edit distance computation stage.

Figure 16 points that for a skewness of less than 10 percent, it is very likely that the computation time elapsed in largest forest is somewhere between 30-40 percent of the final edit distance time. For inputs with higher skewness the fraction of time elapsed in forest distance computation increases. Workloads in which skewed inputs are the common case and skewness is obstructing the performance goals, then directed experiments will be needed. This could be especially needed in case target platform is resource constraint.

7. Discussion

Platform-centric data shaping is a potent mechanism to address the data- and architecture-borne issues in context of semi-structured processing on GPU and modern computing platforms. The platform-centric data shaping techniques developed in this work and elsewhere (Shukla 2017) are relevant to a variety of computing platforms ranging from traditional multi-core CPUs to GPGPUs and accelerators, to SoCs.

Our study point several interesting aspects of compute requirements in various stages of parallel tree edit distance problem. Based on the research conducted in this work, we learn the followings w.r.t. key stages of tree edit distance computation problem on parallel hardware:

1. Partial edit distance computation: GPUs are preferred for computing partial edit distance values, especially, when input trees are large, for example 300 nodes or more. As far as CPUs and SoC are concerned, the cores are a good choice. Reason being that with SoC as target platform, small number of cores, in the range of 4-8, are mostly enough to get any meaningful speedup for the partial edit distance stage. However, graphics core should be preferred for partial edit distance computation in SoC settings.

2. Final edit distance stage: Regarding pattern of compute requirements in final edit distance computation stage which help decide appropriate compute substrate. Based on degree of skewness of input trees, decision to map the computation of specific stages to specific type of compute substrate. Specifically, if trees are on larger side, say 1000sh nodes, and there is a skewness in input, the computation of largest (and k-largest) forest distance be offloaded graphic cores on SoC. This offloading decision is possible at the compile time itself by providing appropriate hints in the code. However, if number of nodes involved are on lower side, for example, less than 300, offloading the largest (k-largest) forest distance computations to graphics cores shall not result in any meaningful speedup. Therefore, no offloading shall be avoided matter even if largest forest distance computation amounts for a large fraction of final edit distance time. The SoC architecture offers another advantage in context of final edit distance computation: small number of cores, 200sh graphic cores, on chip L2 and shared memory, in conjunction with Shared Virtual Memory (SVM) (Intel, 2015) technology is very amenable to the mixed computing requirements for this particular stage.

8. Related Work

Several tree mining methods have been proposed for finding similarities in context of ordered and unordered trees (Bille 2005).

Typically, the similarity-based methods compare a pair of trees using nodes and paths of two trees, and estimate the similarity between them (Zhang and Shasha 1989). A similarity-based approach exploiting the level information of tree nodes is proposed (Nayak 2008). The basic idea is to consider common nodes (of the trees) in the corresponding levels and assigning different weight to different levels (Nayak 2008). A Tensor Space Model (TSM) is used for representing tree data and finding similarities between a pair of trees (Cai et al. 2006). Techniques such as TSM have drawbacks of high dimensionality and complexity (Kutty et al. 2010).

The tree edit distance (TED) based methods have also been proposed in context of unordered trees. These
methods exhibit NP-hard complexity [Zhang et al. (1992), Hirata et al. (2011). Some studies address the complexity issue involved with computing tree edit distance for unordered trees. Researches in [Fukagawa et al. (2011), Mori et al. (2012] reduce the tree edit distance problem to the maximum clique problem. And a variant of the tree edit distance problem is proposed [Torsello and Hancock (2003)]. However, these methods are not appropriate for large unordered trees, as they suffer from high complexity [Fukagawa et al. (2011).

There also exist other methods for matching unordered trees such as maximum agreement subtree [Amir and Keselman (1997), Cole et al. (2000], largest common subtree [Akutsu and Halldorsson (2000), and smallest common supertree [Gupta and Nishimura (1998). These methods also suffer from the high computational complexity. For more related work see Section Background.

9. Conclusions

Big Data analytics in near real-time is becoming important for several objectives. Tree matching is a core component for many applications such as fraud detection, spam filtering, information visualization and extraction, user authentication, natural language processing, XML databases, bioinformatics, etc. Comparing ordered (unordered) trees is compute-intensive, in particular for Big Data. Comparing ordered (unordered) trees is both compute- and data-intensive. In order to facilitate comparison of ordered trees, we have addressed the following problem: How to shape the semi-structured data to enable time efficient processing on a parallel hardware? Specifically, our data shaping approach enables pre-computation of partial edit distance values in parallel. We also develop processor-specific techniques keeping in mind compute requirements of various constituent stages of PTED.

We evaluate our work using real world data sets. Our experimental results show that our SIMT-based PTED-GPU implementation shows speedup of up to 12X when compared with the state-of-the-art in tree edit distance (TED) computation. In addition, our techniques when ported on CPUs are scalable. Finally, we discuss appropriateness of compute platforms w.r.t. various constituent stages of proposed PTED.

10. Future Research

From this work, we learned that data shaping technique helps when there is a higher proportion of leaf nodes in the tree. One limitation of our approach is that a post-processing step is needed. As a part of our future work we would like to reduce the overhead incurred in this step.

This study points towards several other interesting research directions. We understand that changes to structure of trees, for example, orientation of subtree, can result in different computation patterns. This idea can be used to design a efficient techniques for tree matching problems in context of unordered trees and for problems involving tree rotations common in bioinformatics. Another possible direction is to explore data shaping-based solution to regular trees, for e.g., right-only trees, left-only trees, binary trees, or regular k-ary trees.

ACKNOWLEDGEMENT The research reported in this paper is in part supported by Jerry R. Junkin’s Chair and Philip and Virginia Sproul Professorship Endowments at Iowa State University, Ames, IA. The research reported in this paper is, in part, also supported by the HPC@ISU equipment at Iowa State University, some of which has been purchased through funding provided by NSF under MRI grant number CNS 1229081 and CRI grant number 1205413, and IBM Faculty Award.

REFERENCES


Cai, Deng, He, Xiaofei and Han, Jiawei (2006). Tensor space model for document analysis , , 625-626, 2006.


Kouylekov and Magnini (2005). Recognizing textual en-

Kutty, Sangeetha, Nayak, Richi and Li, Yuefeng (2010). XML Documents Clustering Using Tensor Space Model--A Prelimi-


M. Kouylekov and B. Magnini (2005). Recognizing textual en-


Zhang, Kaizhong and Shasha, Dennis (1989). Simple fast algo-


Authors:

Parijat Shukla: Parijat Shukla graduated from Iowa State University with PhD in Com-
puter Engineering. He was a member of Dep-
dependable Computing and Networking Lab (DCNL) directed by Prof. Somani. His re-
search interests are in computer systems ar-
chitecture, high performance computing, Big Data, and fault tolerant computing.

Arun K. Somani: Arun Somani is cur-
cently Distinguished Professor of Electrical and Computer Engineering at Iowa State Uni-
versity. His research interests are in de-
sign and implementation of scalable and de-
pendable computing and networking system,
WDM-based optical fiber networks, and par-
allel computer system architecture, applica-
tions of sensors in critical infrastructure pro-
tection, and image-based navigation. Profes-
sor Somani has served in many leadership positions, several program
committees of various conferences, as IEEE distinguished visitor, IEEE
distinguished tutorial speaker, IEEE Communication Society Disting-
guished Lecturer, and has delivered several keynote speeches, tutorials
distinguished and invited talks all over the world. He has served as Editor of several prestigious IEEE and Elsevier Journals. Professor Somani is a fellow of IEEE, AAAS, and Distinguished Engineer of ACM.