Suffix Tree Construction Algorithms on Modern Hardware

Dimitris Tsirogiannis
University of Toronto
Toronto, Canada
dimitris@cs.toronto.edu

Nick Koudas
University of Toronto
Toronto, Canada
koudas@cs.toronto.edu

ABSTRACT

Suffix trees are indexing structures that enhance the performance of numerous string processing algorithms. In this paper, we propose cache-conscious suffix tree construction algorithms that are tailored to CMP architectures. The proposed algorithms utilize a novel sample-based cache partitioning algorithm to improve cache performance and exploit on-chip parallelism on CMPs. Furthermore, several compression techniques are applied to effectively trade space for cache performance.

Through an extensive experimental evaluation using real text data from different domains, we demonstrate that the algorithms proposed herein exhibit better cache performance than their cache-unaware counterparts and effectively utilize all processing elements, achieving satisfactory speedup.

Categories and Subject Descriptors

H.3.1, H.3.4 [**INFORMATION STORAGE AND RETRIEVAL**]: Content Analysis and Indexing, Systems and Software.

General Terms

Suffix tree, Multi-core, Performance

1. INTRODUCTION

Suffix trees are indexing data structures that allow for efficient computation of many string operations such as exact string matching and the longest common substring problem [14]. Since suffix trees advance the performance of many string processing operations, they are ideal data structures for a wide range of domains in which text data are processed. For example, in bioinformatics, suffix trees have been utilized to effectively search for patterns in databases of genomic DNA data [3]. The full characterization of the human genome exploded the size of genome data stored[1], thereby necessitating the development of efficient text indexing techniques.

Text data generation is occurring at unprecedented rates in the web, corroborated by user generated content in the form of blogs and other social media. Exploiting this vast amount of collective knowledge requires efficient text indexing techniques. Traditionally, inverted indices have been utilized to index text collections. A major limitation of inverted indices is that they consider text to be a sequence of words. Hence, they cannot efficiently answer complex queries such as phrases which are commonly used in the context of online advertising [18]. In contrast, suffix trees enable searching for phrases at a cost that does not depend on the size of the text collection.

Due to the evident importance of suffix indexing data structures, several in-memory suffix tree construction algorithms have been proposed in the past [13, 21, 27, 28]. However, these approaches have not considered the characteristics of modern hardware architectures, namely multi-level cache hierarchies and on-chip parallelism, thus severely underutilizing hardware resources. This shortcoming did not draw particular attention in the past, because the large space requirements of suffix trees had shifted the focus towards the design of algorithms optimized for disk performance, which utilized in-memory algorithms as building blocks [2, 7, 12, 16, 22, 26]. Nevertheless, increasing memory sizes enhance the impact of in-memory tasks on performance. Hence, it is imperative to reassess the performance of in-memory suffix tree construction algorithms and to propose new algorithms that incorporate the characteristics of modern hardware architectures, such as multi-level memory hierarchy and chip multiprocessors.

1.1 Multi-level Memory Hierarchy

The advent of 64-bit hardware architectures removed the 4GB memory size barrier, enabling systems to deploy hundreds of GBs of main memory[2] and many operations can be performed now entirely in memory. A critical drawback of increasing memory size is that it also increases memory access cost. That cost is intensified due to TLB (translation look-aside buffer) misses during the translation of logical to physical memory addresses. To reduce the memory access bottleneck, hardware engineers introduced multi-level memory hierarchies containing two or three levels of cache memories.

The multi-level memory hierarchy results in non-uniform memory access cost. Accessing the L1 cache (closest to the

[1]The current size of the GenBank database is 390GB.

[2]http://www.sun.com/servers/

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EDBT 2010, March 22–26, 2010, Lausanne, Switzerland.
Copyright 2010 ACM 978-1-60558-945-9/10/0003 ...$10.00

CPU) costs 1-3 CPU cycles, while the cost of accessing the L2 cache is 10-20 cycles [9]. A random memory access is an order of magnitude more expensive, requiring hundreds of CPU cycles.

The non-uniform memory access cost necessitates the design of in-memory algorithms that are optimized for cache performance. Although several efforts have been made to design *cache-aware* query processing and indexing techniques [6, 23], suffix tree construction algorithms have not yet been optimized for cache performance.

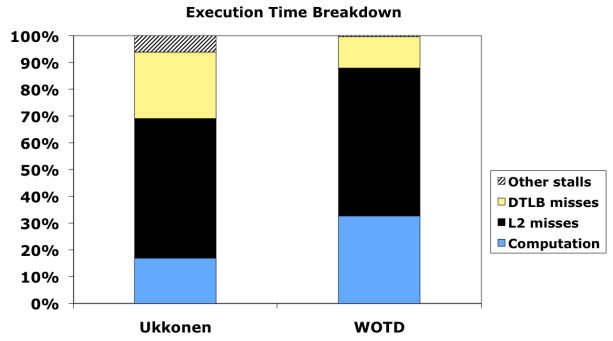

Figure 1: Execution time breakdown of two suffix tree construction algorithms.

To quantify this limitation, we analyzed the execution time breakdown of two widely used suffix tree construction algorithms for an input string of size 100MB (DNA sequence). Using Intel's VTune performance analyzer, we measured how much time each of these algorithms spends on useful computation and how much time is wasted on stalls like cache and TLB misses. As illustrated in Figure 1, all the algorithms examined spend more than 70-80% of their time "waiting" for data to arrive from memory and on other stalls. Hence, there is a significant potential for performance improvement.

1.2 Chip-Multiprocessors

In a continuous effort to improve CPU performance and overcome physical limitations such as heat dissipation, CPU designers introduced the concept of on-chip parallelism or *chip-multiprocessor* architectures (CMPs). The placement of multiple CPUs (*cores*) on the same die has been enabled by higher levels of integration, allowing the concurrent execution of multiple threads of control. Unlike *shared memory* architectures, in a CMP the cores share part of the cache hierarchy – the size of which is in the order of few MBs – the memory bandwidth and the system bus. Thus, in order to exploit the computational power of CMPs, algorithms must be carefully designed so that cores are not competing for shared resources.

In the recent past, there has been a continuous effort to exploit CMP architectures in order to improve the performance of computationally intensive tasks [8, 11]. However, existing suffix tree construction algorithms have not considered the potential performance improvement that CMPs offer.

1.3 Contributions

In this paper, we study the problem of improving the performance of in-memory suffix tree construction algorithms by incorporating the characteristics of modern hardware architectures. In particular, our goals are the following: a) to improve cache performance, and b) to exploit on-chip parallelism of modern CMP architectures.

Initially, we consider the problem of *cache-partitioning* in the context of suffix trees. Cache-partitioning provides the means to parallelizing a task and improving its cache performance. As we demonstrate later, existing partitioning techniques introduce significant overhead that negates the benefits of improved cache performance. Hence, we propose a low overhead cache-partitioning algorithm, termed *PreCache*, that is utilized as a building block in suffix tree construction algorithms.

Next, we propose two suffix tree construction algorithms, termed *CMPUTree* and *MAPST* (**MA**terialized **P**refixes **S**uffix **T**ree), which are tailored to CMP architectures. *CMPUTree* is a cache-conscious parallel adaptation of Ukkonen's algorithm [27]. Ukkonen proposed a linear time suffix tree construction algorithm that severely underutilizes modern hardware resources. In contrast, *CMPUTree* is a linear time algorithm that utilizes Ukkonen's algorithm and the *PreCache* partitioning technique as building blocks in order to construct suffix trees in a cache-conscious and parallel way.

The second algorithm proposed, termed *MAPST*, is a cache-conscious suffix tree construction algorithm tailored to CMPs that is more space efficient than *CMPUTree* at the expense of a theoretical worse construction time ($O(n \log n)$). Inspired by the *WOTD* algorithm [13], *MAPST* constructs the suffix tree in a top-down fashion, allowing for a more space efficient representation of tree nodes. By utilizing materialized prefixes as well as different compression techniques, it effectively trades space for cache performance, significantly reducing the suffix tree construction cost. Furthermore, it employs *PreCache* as a building block to parallelize the suffix tree construction task and to bound the size of each core's working set.

Finally, we present an extensive experimental evaluation of the algorithms proposed herein using real text corpora from different domains. We demonstrate that the applied techniques improve cache performance of in-memory suffix tree construction algorithms. Furthermore, all the algorithms proposed achieve satisfactory speedup as we increase the number of cores.

The remainder of this paper is organized as follows. An extended related work as well as background information on suffix trees are presented in Sections 2 and 3, respectively. In Section 4, we describe the cache-partitioning algorithm (*PreCache*) that is utilized as a building block in the suffix tree construction algorithms (*CMPUTree* and *MAPST*), presented in Section 5. We experimentally demonstrate the efficacy and efficiency of the proposed algorithms in Section 6 and we conclude in Section 7.

2. RELATED WORK

In this section, we describe existing suffix tree construction algorithms by categorizing them as follows: a) in-memory algorithms (Section 2.1), b) algorithms that are optimized for disk performance (Section 2.2), and c) parallel algorithms (Section 2.3). This is by no means an exhaustive list of existing approaches; due to space constraints, we only consider those that are most relevant to our work.

2.1 In-memory Algorithms

Linear time algorithms for constructing suffix trees in main memory have been proposed by Weiner [28], McCreight [21], and Ukkonen [27]. One common characteristic of these algorithms is that, due to the use of suffix links, they all have irregular tree traversal patterns and hence, they exhibit poor cache performance. However, Ukkonen's is the algorithm of choice due to the following reasons: a) it poses less space requirements in practice, b) it is constructed online, and c) it is easier to understand and implement. A high level description and the limitations of Ukkonen's algorithm are presented in detail in Section 3.

Several algorithms have been proposed that abandoned the use of suffix links in order to reduce the space requirements of suffix trees and improve temporal locality [7, 13]. These algorithms exhibit an $O(n^2)$ worst time complexity; $O(n \log n)$ for finite alphabets. The most space efficient suffix tree construction algorithm in this category is the write-only top-down algorithm (*WOTD*) proposed by Giegerich et al. [13]. The *WOTD* algorithm proceeds in a top-down fashion, completely evaluating each tree node (identifying child nodes and edge labels) before it proceeds to the next one. Hence, it is able to apply a more space efficient representation of tree nodes. Furthermore, the child nodes of a particular tree node are stored in contiguous memory locations, resulting in better cache performance during query time. Nevertheless, *WOTD* exhibits poor temporal locality during the evaluation of tree nodes and it is not designed for CMP architectures.

Other linear time algorithms have also been proposed in [10, 17]. Karkkainen and Ukkonen introduced the notion of *sparse suffix trees (SST)* to represent a subset of suffixes and proposed a linear time construction algorithm for sparse SSTs [17]. Farach et al. proposed a linear time in-memory suffix tree construction algorithm for integer alphabets [10]. However, these algorithms have not been optimized for cache performance.

2.2 Disk-based Algorithms

Existing in-memory algorithms exhibit random access patterns which renders them ill-suited for disk-based scenarios. To address these issues, many suffix tree construction algorithms have been proposed that are optimized for disk performance [2, 7, 12, 16, 22, 26].

Hunt et al. proposed a disk-based suffix tree construction algorithm for biological sequences [16]. Instead of using suffix links, their algorithm performs multiple passes over the input string in order to construct the suffix tree. In every pass, the suffixes with a specific prefix are indexed. Bedathur and Haritsa proposed a low-overhead buffering policy, called TOP-Q, to improve the on-disk behavior of suffix tree construction algorithms [2]. Cheung et al. proposed a top-down disk-based suffix tree construction algorithm, termed *DynaCluster*, that utilizes a fixed-length prefix-based partitioning technique [7].

A disk-based suffix tree construction algorithm, termed *TDD*, that utilizes *WOTD* as building block was proposed by Tian et al. [26]. TDD applies a different buffer replacement policy for every data structure of *WOTD*. A fixed-length prefix-based partitioning technique is employed to divide the suffix tree into independent sub-trees such that each sub-tree is constructed in memory. A variant of this algorithm, termed *ST-merge*, is proposed for the case when the input string does not fit in memory.

Phoophakdee and Zaki proposed a disk-based suffix tree construction algorithm that utilizes Ukkonen's algorithm [22]. Their algorithm, termed *TRELLIS*, deploys a variable-length prefix-based partitioning algorithm to partition the suffixes of the input string into groups such that the suffix tree of each group fits in main memory. The input string is initially divided into a number of consecutive and disjoint substrings and a suffix tree is constructed from each substring in main memory. The sub-trees are subsequently merged into a single suffix tree in a disk optimized way.

Ghoting and Makarychev proposed serial and parallel suffix tree construction algorithms to deal with the case where the input string does not fit in memory [12]. By carefully adapting a simple suffix tree construction algorithm, their algorithm, termed *Waterfront*, is able to maintain a constant working set size, effectively reducing the amount of I/O performed. A parallel version is proposed that is tailored to massively parallel shared-nothing architectures.

2.3 Parallel Algorithms

Theoretical parallel suffix tree construction algorithms have been proposed in the past [15, 20]. The algorithm in [20] runs in $O(\log n)$ parallel time and uses n processors. The first work-optimal parallel suffix tree construction algorithm was presented in [15]. It does $O(n)$ work and runs in $O(\log^4 n)$ time using n processors. To the best of our knowledge, no practical implementations of these algorithms have been reported.

A parallel algorithm for constructing suffix trees on a computational grid was proposed by Chen and Schmidt [5] in which the suffixes are partitioned into groups using a fixed-length prefix-based partitioning technique. A greedy load balancing algorithm is applied to assign the construction of each suffix tree to a processing node and a suffix tree is constructed from each group utilizing Ukkonen's algorithm.

Carvalho et al. studied the problem of determining a balanced partition of a lexicographic trie in the context of parallelizing the extraction of structured motifs and proposed an approximate partition algorithm with explicit load balancing guarantees [4]. However, their algorithm cannot be utilized for cache-partitioning as it does not provide any guarantees with respect to the size of the partitions produced.

3. BACKGROUND

In this section, we present a general description of suffix trees. We also provide a high level description of Ukkonen's algorithm as it is used as building block in *CMPUTree* (Section 5).

3.1 Suffix Trees

Let Σ be an alphabet of $|\Sigma|$ symbols. We denote $S = s_1, \ldots, s_n\$$ to be a string over Σ of length $n \geq 1$, where $s_i \in \Sigma$ and \$ is a terminating symbol not occurring in S. For any $i \in [1, n]$, $S_i = s_i \ldots s_n$ denotes the i-th suffix of S ($S_1 = S$). A suffix tree of S, $ST(S)$, is a rooted tree with the following properties. Every node, except from the root, has at least two edges and every edge has a label representing a substring of S. For every node, each of its edges starts with a different symbol from Σ. For every node u, $p(u)$ denotes the path from the root node to u and $\overline{u}$ is the concatenation of edge labels on the path from the root to u. Every leaf node corresponds to a suffix of S. Finally, a pointer from a

node u to another node v is called *suffix link* if $\overline{u} = \alpha x$ and $\overline{v} = x$, where $\alpha \in \Sigma$ and x is a substring of S. The suffix tree of string $BANANA\$$ is presented in Figure 2.

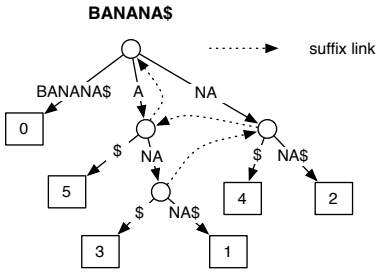

Figure 2: Suffix tree of string BANANA$.

3.2 Ukkonen's Algorithm

Ukkonen's algorithm constructs the suffix tree of a string S in n *phases*, where n is the size of S [27]. It processes the input string from left to right and during each phase it expands the partially constructed suffix tree by considering a larger prefix of S. Algorithm 1 presents a high-level description of Ukkonen's algorithm.

Algorithm 1 *Ukkonen's Algorithm*

Require: The input string $S = s_1, \ldots, s_n$ of size n
Ensure: The suffix tree of S
1: Initialize an empty suffix tree.
2: **for** i from 1 to $n-1$ **do**
3: Begin *phase* $i+1$.
4: **for** j from 1 to $i+1$ **do**
5: Locate in the current tree the path with the longest common prefix with substring $s_j, \ldots, s_i$.
6: **if** symbol s_{i+1} is not already present at the end of that path **then**
7: "extend" the path by adding symbol s_{i+1} to ensure that substring $s_j, \ldots, s_{i+1}$ is in the tree.
8: **end if**
9: **end for**
10: **end for**

The time complexity of Algorithm 1 is $O(n^3)$. Ukkonen made several observations that enabled the reduction of time complexity to $O(n)$. In this section, we discuss those that are of interest to this paper. The first observation was that edge labels do not have to be explicitly stored in a suffix tree. Instead, a pair of string indices is sufficient for representing the corresponding substring. The indices specify the beginning and end position of a substring in S.

Secondly, Ukkonen observed that traversing the partially constructed suffix tree to locate the path that has the longest common prefix with the currently examined substring is a frequent and expensive operation. Ukkonen utilized suffix links to reduce the number of edges examined in this step and consequently, the algorithm's time complexity.

Despite its linear time complexity, Ukkonen's algorithm has a number of major limitations in practice. Firstly, it exhibits poor cache performance which is attributed to two reasons: a) the irregular tree traversal, and b) the implicit representation of edge labels using indices. Secondly, Ukkonen's algorithm is not space efficient and exhibits poor cache performance during query time. Even with the most space efficient implementations of suffix trees [19], Ukkonen's algorithm requires at least 20 bytes per symbol on a 32-bit architecture.

4. CACHE-PARTITIONING

Partitioning is the process of dividing data into *partitions* so that a particular task can be performed as a set of subtasks, each applied to a different partition. Depending on the role of partitioning, different requirements are imposed. For example, if the goal is to improve disk performance, the requirement is that every sub-task is performed on memory-resident data. Similarly, when the goal is to improve cache performance (*cache-partitioning*), the working set of each sub-task is required to fit in cache. The working set of each sub-task must fit in the $1/C$-th part of the cache when a CMP with C cores is considered, otherwise the cores start competing for the shared resource.

Cache-partitioning is more challenging than partitioning for disk performance. The performance improvement from reducing disk accesses compensates by a considerable margin the overhead of the applied partitioning technique and the partitioning cost has a minor contribution to overall execution time [22]. This is not the case for cache-partitioning. Dividing large datasets into many small (in the order of few MBs) partitions such that all partitioning requirements are met may incur significant overhead that negates the benefits of improved cache performance [25].

In the context of disk-based suffix tree construction algorithms, a prefix-based partitioning technique is typically employed to distribute the suffixes of a string into a number of memory-sized partitions such that every partition contains suffixes with the same prefix; variable-length prefixes are utilized to handle skewed data. As we demonstrate in Section 6, these techniques are ill-suited for cache performance as they introduce notable overhead.

4.1 PreCache Partitioning Algorithm

In this section, we present *PreCache*, a low overhead partitioning algorithm that divides the suffixes of a string into a number of cache-sized prefix-based partitions. *PreCache* is tailored to CMPs and is highly efficient for the following reasons: a) it utilizes all processing elements (cores), b) it exploits the shared L2 cache and the fast inter-processor communication on CMPs to improve cache performance and reduce the demand for memory bandwidth, and c) it employs sampling to reduce partitioning cost.

For a partition p, $freq(p)$ denotes the number of suffixes of p. Formally, we define the partitioning problem as follows. Given a string S of size n, compute a set of prefix-based partitions P such that the following conditions are satisfied: a) every suffix belongs to exactly one partition (*coverage condition*), b) $\forall P_i \in P : freq(P_i) \leq th$ (*space condition*), and c) $\forall P_i \in P$, all the suffixes of P_i have prefix $\overline{P_i}$ in common; in this case, partition P_i *corresponds* to prefix $\overline{P_i}$.

The *PreCache* algorithm proceeds in two steps. The partitions of P as well as their corresponding prefixes are computed in the first step. The partitions are populated with the suffixes of the input string in the second step. In the following sections, we describe these two steps in detail. In Section 5, we also describe how the space condition is set in order to improve the cache performance of suffix tree construction algorithms.

4.1.1 Computing the Partition Set

Partition *expansion* is an essential procedure of the *PreCache* algorithm. *Expanding* a partition p, that corresponds to a prefix of size l, is the process of replacing p with $|\Sigma|$ new – potentially smaller – partitions $p_1, \ldots, p_{|\Sigma|}$ such that every partition p_i corresponds to a distinct prefix of size $l+1$. For example, if we assume an alphabet $\Sigma = \{A, C, G, T\}$, expanding a partition p, where $\overline{p} = A$, would result in the replacement of p with four new partitions $\overline{p_1} = AA$, $\overline{p_2} = AC$, $\overline{p_3} = AG$, $\overline{p_4} = AT$.

Algorithm 2 PreCache - Partition Set Computation

Require: S: string, l: initial prefix length, th: threshold, B: number of substrings of length l_{cur} in the sample of S
Ensure: P: partition set
1: Set $l_{cur} \leftarrow l$.
2: Initialize a partition set P' with $|\Sigma|^l$ partitions.
3: Compute a sample of S. // The sample units are substrings of S.
4: **while** P' is not empty **do**
5: **for** each sample unit s **do**
6: **for** every substring sb of s of length l_{cur} **do**
7: **if** $sb \in P'$ **then**
8: Increment the counter of the corresponding partition.
9: **end if**
10: **end for**
11: **end for**
12: **for** every partition p of P' **do**
13: Set $\overline{freq}(p) = count(p) \cdot n/B$
14: **if** $\overline{freq}(p) > th$ **then**
15: ***Expand*** p into $p_1, \ldots, p_{|\Sigma|}$ and insert them into P'.
16: **else**
17: Remove p from P' and insert it into P.
18: **end if**
19: **end for**
20: Set $l_{cur} = l_{cur} + 1$. // Increment l_{cur}
21: **end while**

Given the *expansion* procedure, the computation of the prefix-based partitions works as follows (presented in Algorithm 2). *PreCache* creates an initial set of partitions P' (line 2) that cover all suffixes of S and progressively computes the final partition set P by performing multiple passes over a sample of S (lines 4-21). In every pass, the partitions that satisfy the space condition are removed from P' and stored in P (line 17). The partitions of P' that do not satisfy the space condition are *expanded* into $|\Sigma|$ new partitions which are inserted in P' (the original partition is discarded from P') (line 15) and a new pass over the sample of S is performed. Algorithm 2 terminates when all the conditions are met, i.e. P' is an empty set.

A high level description of Algorithm 2 is presented in Figure 3. In this particular example, three passes are required to divide the suffixes of the input string into a number of partitions such that the space requirement is satisfied, i.e. the number of suffixes of each partition is less than or equal to two. The numbers next to the prefixes indicate the number of suffixes of each partition. Next, we describe in greater detail the most important implementation aspects of Algorithm 2.

P' is initialized with the partitions that correspond to all possible prefixes of length l, i.e. the size of P' is $|\Sigma|^l$ (line 2). P' is stored in a hash table and the value of l is set so that P' fits in cache. For every partition of P', the corresponding prefix and a counter are stored in a hash bucket.

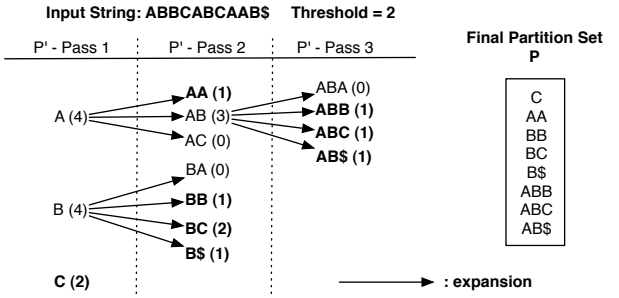

Figure 3: Example of variable-length prefix-based partitioning.

The address of the hash bucket is determined by applying a hash function on the prefix. Since sampling is utilized, computing the exact number of suffixes of each partition is impossible. Instead, a frequency estimate ($\overline{freq}$) is computed for each partition of P' in every pass (lines 5-11). The frequency estimate of a partition is stored in the counter of the corresponding hash bucket.

In every pass, the frequency estimates are computed from a random sample of S that consists of a number of sample units (substrings of S), each containing b symbols. We set the value of b to be a multiple of a cache line size (typically 64 or 128 bytes) and we align sample units to cache line boundaries in order to improve spatial locality and memory bandwidth utilization. From every sample unit, we extract all possible substrings of length l_{cur} to probe the hash table that stores P'. If a substring is found in the hash table, we increment its counter (lines 7-9). If this is not the case, the substring is ignored. Once all the sample units have been processed, we scale the counter of each partition p by n/B to get a frequency estimate (line 13) and we compare this value with th (threshold) to decide whether to *expand* p or place it in P (lines 14-18); B is the number of substrings of size l_{cur} in the sample of S.

Similar to hash-based aggregation on CMPs, there are multiple ways to execute Algorithm 2 that either introduce significant synchronization overhead (single shared hash table) or increase memory footprint (multiple hash tables) [8]. In this work, we apply a different approach that avoids the overhead of synchronization and does not increase memory requirements at the expense of extra CPU overhead. A single hash table is utilized and its hash buckets are partitioned to cores such that every core updates the frequencies of a disjoint set of hash buckets. The pitfall of this approach is that every core has to process the entire sample of S and discard those substrings that are not hashed to the set of assigned buckets. However, there is a great advantage when this approach is applied to a CMP. Since the same input is processed by all cores, only one core (the fastest) reads the sample units from memory, operating as a prefetcher. The remaining cores exploit the fast inter-processor communication of CMPs to effectively retrieve the sample units from the cache, thus reducing memory accesses and the demand for memory bandwidth.

Once all cores have processed the sample of S, the partitions that satisfy the space condition are stored in P while the remaining partitions are expanded. A single core is responsible for initializing the hash table with the new partitions of P' and for assigning the hash buckets to cores. All

cores proceed to the next phase at the same time, this being the only synchronization point.

4.1.2 Populating the Partitions

Once P has been computed, its partitions are populated with the suffixes of the input string. A single pass is performed over S to identify the partition of each suffix. The partitions of P are stored in a hash table using their prefixes as keys. The payload for each partition is a vector of suffix indices.

To determine the partition of a suffix s, the hash table that stores P is probed using at most $l_{\max} - l$ prefixes of s of sizes $l, l+1, \ldots, l_{\max}$, where $l_{\max}$ is the length of the longest prefix that corresponds to a partition of P. The partition of s is the one corresponding to the smallest of these prefixes for which a match is found in the hash table; the index of s is stored in the corresponding suffix indices vector. Populating the partition set requires $\frac{n}{2} \cdot (l_{\max} - l)$ hash probes in expectation.

For the case of populating the partitions on a CMP, the best approach is to range partition S (divide in consecutive substrings) among cores and deploy a single hash table in conjunction with a synchronization mechanism (mutexes). The "shared input" approach applied in Section 4.1.1 is not optimal in this case for the following reason. Since S is significantly larger than the size of the shared cache, the cores cannot effectively coordinate in order to process the same part of the input string simultaneously. Consequently, they end up competing for the shared cache, resulting in a thrashing behavior. Using a separate hash table per core was also found to be suboptimal due to the increasing number of TLB misses, attributed to a considerably larger number of memory addresses accessed (the number of vectors increase by a factor of C).

5. SUFFIX TREE CONSTRUCTION ALGORITHMS FOR CMP

In this section, we present *CMPUTree* and *MAPST*, two cache-conscious suffix tree construction algorithms tailored to CMP architectures. Initially, we present a framework for exploiting on-chip parallelism that is utilized by both algorithms (Section 5.1). Subsequently, *CMPUTree* and *MAPST* algorithms are presented in Sections 5.2 and 5.3, respectively. The description of the aforementioned algorithms focuses on how to improve cache performance.

5.1 Exploiting on-chip Parallelism

In this section, we present a framework for constructing a suffix tree on a CMP with C cores. The framework comprises four main phases which are executed in the order presented herein: a) *partition phase*, b) *construction phase*, c) *merging phase*, and d) *suffix-links recovery phase*. As we demonstrate in Section 6, the total cost of constructing a suffix tree depends primarily on the time spent on the construction phase. The last two phases are optional as they affect performance during query time.

5.1.1 Partition Phase

In the partition phase, the suffix tree construction task is divided into a number of sub-tasks such that each subtask can be executed independently by a single core. In this phase, the *PreCache* partitioning algorithm (Section 4) is utilized to divide the input string into a number of prefix-based partitions such that a suffix tree can be constructed from each partition in cache. Then, a simple scheduling algorithm is employed to assign these partitions to cores. The scheduling algorithm is the best-fit heuristic algorithm for solving the bin-packing problem [29].

5.1.2 Construction Phase

In the construction phase, every core builds a suffix tree from each assigned partition. *CMPUTree* and *MAPST*, as well as any other in-memory suffix tree construction algorithm can be utilized as a building block in this phase.

5.1.3 Merging Phase

The merging phase is employed to merge into a single suffix tree the suffix trees that were produced in the previous phase. Recall that every suffix tree corresponds to a prefix-based partition. To merge the suffix trees, the corresponding prefixes are considered to be suffixes of a single string and a suffix tree, ST', is constructed from these suffixes. By the definition of suffix trees, the number of leaf nodes of ST' is equal to the number of partitions and every leaf node corresponds to the root node of a suffix tree. Hence, the merging phase is reduced to the construction of ST'.

To execute the merging phase on a CMP with C cores, the prefixes are divided into $|\Sigma|^l$ fixed-length prefix-based partitions, where l is the smallest prefix length satisfying $|\Sigma|^l \geq C$. The partitions are assigned to cores using the same scheduling algorithm as in Section 5.1.1 and every core constructs a part of ST' independently. A single core is responsible for merging these parts and connecting the leaf nodes of ST' to the root nodes of the suffix trees.

5.1.4 Suffix-links Recovery Phase

Since suffix links are needed in many string processing algorithms, we provide an optional suffix links recovery phase. Recovering the suffix links requires a depth-first traversal of the suffix tree. To perform this operation on a CMP, we partition the suffix tree among cores using fixed-length prefix-based partitions and every core computes the corresponding suffix links.

5.2 CMPUTree Algorithm

In this section, we present the *CMPUTree* algorithm, a cache-conscious suffix tree construction algorithm that utilizes Ukkonen's algorithm as a building block. In order to improve cache performance during the construction of a suffix tree, we need to ensure that the working set of Ukkonen's algorithm fits in cache. The working set of Ukkonen's algorithm consists of the partially constructed suffix tree and the substrings referenced by its edges. *CMPUTree* employs the *PreCache* algorithm to divide the input string into prefix-based partitions such that each partition contains at most th suffixes. By properly setting the value of th, we ensure that all memory accesses are resolved through cache during the construction of a suffix tree. In particular, we set the value of th so that the following condition is satisfied:

$$th \cdot (node\_size + cache\_line) \leq \frac{L2}{C},$$

where *node_size* is the size of a tree node (it depends on implementation details), *cache_line* is the cache line size, and $L2$ is the size of the L2 cache.

The product $th \cdot node\_size$ denotes the space requirement of tree nodes and $th \cdot cache\_line$ is the space requirement of edge labels (substrings). We set the space requirement of every referenced substring to be the cache line size for two reasons: a) the cache line is the minimum transfer unit from memory, and b) the length of the longest common prefix and consequently, the length of every edge label is $O(\log n)$ in expectation [1]. Although, there is no explicit guarantee that all edge references will be resolved through cache, we observed that considering the cache line size as the space overhead of each edge label works well in practice (Section 6).

5.3 MAPST Algorithm

In this section, we present the *MAPST* algorithm, a cache-conscious suffix tree construction algorithm for CMPs inspired by *WOTD*. *MAPST* constructs a suffix tree in a top-down fashion, completely evaluating a tree node before proceeding to the next one. Furthermore, it utilizes the same space efficient representation of tree nodes as in *WOTD*. Evaluating a tree node is the process of identifying its child nodes and their corresponding edge labels. In particular, evaluating a node u is performed by processing the set of suffixes that have string $\overline{u}$ as their longest common prefix, a procedure that generates a large number of random memory accesses.

To eliminate the accesses to the input string (suffixes) and improve cache performance during the evaluation of tree nodes, *MAPST* materializes a fixed-length prefix from every suffix. The tree nodes are evaluated using the materialized prefixes without accessing the input string, thus significantly reducing the random memory accesses and the suffix tree construction cost. Different compression techniques are employed by *MAPST*, according to the type of text data, to reduce the space overhead of materialized prefixes and the CPU overhead for evaluating tree nodes. Compression also reduces the probability of accessing the input string as it enables larger prefixes to be materialized in a specific space budget.

Finally, *MAPST* utilizes *PreCache* to ensure that its working set, i.e. the materialized prefixes for constructing the suffix tree of each partition, fits in cache. This is accomplished by setting the value of th such that the following condition is satisfied:

$$th \cdot prefix\_size \leq \frac{L2}{C},$$

where *prefix_size* is the size (in bytes) of each materialized prefix; th is the number of suffixes of each partition. Next, we describe the *MAPST* algorithm in detail.

The pseudo-code of the *MAPST* algorithm is presented in Algorithm 3. We assume that the partitioning phase has already been applied and *MAPST* is utilized to construct the suffix tree from the suffixes of a particular partition. For the evaluation of the root node, *MAPST* produces the initial set of compressed prefixes that are stored in array *Suffixes* (line 1). Counting sort is applied to sort these prefixes based on their first symbol (line 2). After the sorting phase, every symbol of the alphabet corresponds to a (possibly empty) group of suffixes, represented as a range of positions in *Suffixes*. The group corresponding to symbol $\alpha \in \Sigma$ is denoted as α-group.

For example, assume that *Suffixes* contains the following prefixes of length 3: $[AAA, GTA,\ CAT, TTT,\ TTT,\ AGA,\ GAC]$, where $\Sigma = \{A, C, T, G\}$. For simplicity, we do not present the prefixes in compressed form. After sorting the prefixes based on their first letter, *Suffixes* becomes $[AAA, AGA, CAT, GAC, GTA, TTT, TTT]$. Symbol A corresponds to A-group (range $[1, 2]$ of prefixes), symbol C corresponds to C-group (range $[3, 3]$), and so on. The α-groups that are produced during the evaluation of a tree node are stored in a stack (lines 3, 13). A new tree node is created for every α-group that is extracted from the stack and the algorithm terminates when the stack is empty (lines 4-15).

Algorithm 3 *MAPST Algorithm*

Require: S: input string, $p = \{s_1, \ldots, s_{th}\}$: partition with th suffixes, *Suffixes*, *Temp*: arrays of materialized prefixes, Z: compression technique (RLE or LZW), Σ: alphabet
Ensure: ST: suffix tree of the suffixes of p
1: For every suffix of p, produce a compressed prefix using Z and store it in *Suffixes*.
2: Sort the compressed prefixes using the first symbol of each prefix as key. The sorted prefixes are stored in *Suffixes* and every symbol of Σ corresponds to an α-group (range of prefixes in *Suffixes*).
3: Store every α-group in a stack.
4: **while** the stack is not empty **do**
5: Retrieve the next α-group (range $r = [i, j]$ of compressed prefixes) from the stack.
6: **if** r contains a single prefix ($j = i$) **then**
7: Generate a new leaf node in ST.
8: **else**
9: Compute the *lcp* of the suffixes corresponding to the compressed prefixes that are referenced by r.
10: Create a new node in ST and use the *lcp* as the edge label.
11: Update the compressed prefixes of r by removing the *lcp*.
12: Apply counting sort on the updated prefixes of r.
13: Store the α-groups that correspond to the symbols of Σ in stack.
14: **end if**
15: **end while**

If an α-group contains a single prefix, a leaf node is produced (line 7). For instance, when the C-group (represented by range $[3, 3]$) is extracted, a leaf node is created to represent the suffix that corresponds to prefix CAT. An internal node of the suffix tree is produced for every extracted α-group (*group* for simplicity) that contains more than one prefixes (lines 9-13). The edge label of this node is the longest common prefix (*lcp*) of the corresponding suffixes.

The *lcp* is efficiently computed using the materialized prefixes, without accessing the input string (line 9). For example, consider processing the group represented by the range $[1, 2]$ of prefixes. In this case, the *lcp* is computed directly from the cache-resident prefixes, i.e. *lcp(AAA, AGA)* $= A$. However, this may not always be the case. Consider for example the range $[6, 7]$ of prefixes. In this case, we are only certain that the *lcp* contains at least substring TTT. To find the exact *lcp*, the corresponding suffixes must be accessed, resulting in at least two random memory accesses. However, by using compression, *MAPST* materializes larger prefixes in a specific space budget (cache) and reduces the probability of accessing the input string during the evaluation of tree nodes.

Once the *lcp* has been computed from a set of prefixes, these prefixes are updated by removing the *lcp* (line 11). For example, prefixes AAA, AGA of range $[1, 2]$ become AA, GA by removing symbol A. If during that procedure a prefix is

exhausted (no more symbols left), a random memory access is performed to the input string to update the materialized prefix. Again, the merit of using compression is that it reduces the probability of exhausting the materialized prefixes. Counting sort is employed next to sort the resulting prefixes and distribute them to new groups which are inserted in stack (lines 12-13).

Although compression significantly reduces the tree node evaluation cost, it also introduces significant CPU overhead during decompression. Next, we describe two variants of *MAPST* that leverage the "internals" of different compression techniques to effectively reduce the tree node evaluation cost for different types of text data. The following compression techniques are considered: a) run-length encoding (RLE), and b) Lempel-Ziv-Welch (LZW). RLE is appropriate for strings with many sequences of repeated symbols such as biological data. However, for other types of text data that do not have this property, such as English texts, a directory based compression technique such as LZW is more suitable. Both compression techniques are lossless and require a single pass over the input string.

5.3.1 Using RLE

An *RLE-character* is a (α, n) pair, denoting that symbol $\alpha \in \Sigma$ appears n consecutive times. For every suffix s of an α-group we materialize the following fields: a) a compressed prefix of s that consists of k RLE-characters, where k is a user defined parameter, and b) a pointer to s in the input string. For example, for the first suffix (S_1) of string $AAACCCCCAGG$ we would store the pair $(A3C5, 1)$, assuming $k = 2$. Larger values of k enable the evaluation of more tree nodes without accessing the input string. However, the space requirements for storing an α-group increase with k. We demonstrate these tradeoffs in Section 6.

Next, we describe how the RLE encoding is leveraged to reduce the cost of computing the *lcp* of an α-group. For every α-group g, we explicitly store the maximum (max_n) and minimum (min_n) n of the first RLE-character of every suffix of g. We consider three cases with respect to the values of these two variables:

- Case 1: $max_n > min_n$. This is the best case, as we can automatically infer at a cost of a single arithmetic comparison that the *lcp* of g is the single symbol α.
- Case 2: $max_n = min_n > 1$. We know that $lpc \geq \alpha\alpha \ldots \alpha$ (symbol α appears at least max_n times). In this case, we need to examine the materialized prefixes of g to determine the exact *lcp*. However, we do not need to consider the first $maxf$ symbols of each prefix since we already know that they are part of the *lcp*.
- Case 3: $max_n = min_n = 1$. This is the worst case, as we don't have any information regarding the *lcp* and we must examine the prefixes of g exhaustively.

In all but the first case, we have to examine the compressed prefixes in order to compute the *lcp* of g. However, this procedure is highly efficient for the following reasons: a) the compressed prefixes reside in cache, b) the prefixes are stored in contiguous memory locations and are accessed sequentially, and c) the comparisons are performed on RLE-characters, thereby reducing the computational cost. The above are verified in Section 6.

Once the *lcp* of an α-group has been computed, the prefixes are updated by removing the *lcp* and then sorted using counting sort. Both operations can be performed directly on compressed prefixes.

5.3.2 Using LZW

LZW is an efficient dictionary-based compression technique. Commonly, the dictionary, the size of which is user defined, is stored in an *LZW-trie*, where every trie node corresponds to a dictionary entry. We refer to a dictionary entry as *LZW-code*. When LZW is employed in *MAPST*, for every suffix of an α-group we store the following: a) a compressed prefix that consists of k LZW-codes, and b) a pointer to the position of the suffix in the input string.

Computing the *lcp* and sorting the suffixes of an α-group is performed at low cost by accessing the LZW-trie. In *MAPST*, we set the size of the dictionary to be significantly smaller than the cache (in the order of few KBs). The small size of the LZW-trie and the fact that it is frequently accessed, increase the probability of it residing in cache during the construction of the suffix tree.

Algorithm 4 *Algorithm for computing the lcp*

Require: g: α-group, m: number of suffixes represented in g, $P_1, \ldots, P_m$: compressed prefixes of g
Ensure: *lcp* of the suffixes of g
1: $lcp \leftarrow P_1$
2: **for** $i = 2, \ldots, m$ **do**
3: $lcp \leftarrow compute\_lcp(lcp, P_i)$
4: **if** *size(lcp) == 1* **then**
5: Terminate
6: **end if**
7: **end for**

Algorithm 4 presents the pseudo-code for computing the *lcp* of an α-group. The *lcp* is initialized with the first prefix (line 1) and is progressively refined as more prefixes are processed (line 3). If the *lcp* is reduced to a single symbol during this refinement, Algorithm 4 terminates (line 5).

We illustrate Algorithm 4 with the following example. Assume that the *lcp* is initialized with compressed prefix $P_1 = C_1C_2C_3$, where C_1, C_2, C_3 are dictionary entries and consider the process of refining the *lcp* by computing the longest common prefix between the current value of *lcp* (P_1) and the second prefix $P_2 = C'_1C'_2C'_3$. To compute the *lcp* between any pair of compressed prefixes, we initially inspect their code values to efficiently identify a common prefix. Subsequently, we compute the exact longest common prefix by accessing the LZW-trie.

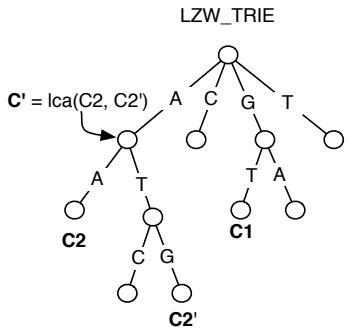

Figure 4: LZW-trie

If we assume in our example that $C_1 = C_1'$ and $C_2 \neq C_2'$, the *lcp* between P_1 and P_2 contains at least C_1 and thus, we only need to compute the *lcp* between C_2 and C_2'. Notice, that we do not need to consider codes C_3 and C_3'. Since every code corresponds to a trie node, computing the longest common prefix is equivalent to computing the lowest common ancestor (*lca*) of the corresponding trie nodes. Hence, we employ a constant-time *lca* retrieval algorithm, proposed in [24].

In our example, the *lcp* between C_2 and C_2' is the trie node that corresponds to code C' (Figure 4) and the new value of *lcp* is C_1C'. The same refinement procedure is then applied to the remaining prefixes as illustrated in Algorithm 4 (line 2) until either the *lcp* contains a single symbol or all the prefixes have been processed. If at some point a prefix is exhausted, the corresponding suffix is accessed and a new compressed prefix is produced. Once the *lcp* has been computed, the prefixes are updated by removing the *lcp* and they are sorted using counting sort. All these operations are performed at low cost by accessing the LZW-trie which with high probability resides in cache.

6. EXPERIMENTAL EVALUATION

In this section, we present the experimental evaluation of the proposed suffix tree construction algorithms using real text corpora. In Section 6.1, we describe implementation details of our algorithms, the datasets used to assess their performance, and the hardware configuration of the CMP on which we run our experiments. The results are presented in Sections 6.2 and 6.3.

6.1 Experimental Setting

We compare the two suffix tree construction algorithms proposed (*MAPST*, *CMPUTree*) against *WOTD* and Ukkonen's algorithm, respectively. Implementations of the latter were retrieved from publicly available sources[3][4]. Initially, the implementation of Ukkonen's algorithm utilized sibling lists (linked lists) to store the child nodes of each tree node. We altered its implementation using a combination of hash map and sibling lists to reduce the number of random memory accesses during the evaluation of tree nodes.

Two factors affect the cost of constructing a suffix tree besides the size of the input string: a) the alphabet size, and b) the average longest-common-prefix. To demonstrate the effectiveness of the proposed suffix tree construction algorithms on datasets with different characteristics, we selected two text collections from Ferragina and Navarro's *Pizza&Chilli* corpus[5]. The first text collection has small alphabet size (16 symbols) and contains gene DNA sequences from the Human Genome Project. The second collection has a relatively larger alphabet size (239 symbols) and is a concatenation of English text files from the Gutenberg Project[6]. The characteristics of the selected collections are summarized in Table 1.

All the experiments were conducted on a Dell PowerEdge 2950 server with two quad-core Intel E5355 CPUs (8 cores) running at 2.6GHz. Each quad-core has a 4MB of shared L2 cache (8MB is total), and every core has a private 64KB L1 cache. The server runs the 64-bit Linux operating system with kernel 2.6.15 and has 32GB of main memory. All the algorithms proposed were implemented in C using Posix threads (Pthreads) and the gcc-4.3 compiler was used with optimization level 02. In all the experiments conducted, the *sched_setaffinity* system call of the Linux OS was employed to schedule the threads to specific cores and ensure that they were not re-scheduled during execution.

[3]http://www.cs.ucdavis.edu/gusfield/strmat.html
[4]http://bibiserv.techfak.uni-bielefeld.de/wotd
[5]http://pizzachili.dcc.uchile.cl/index.html
[6]http://www.gutenberg.org/wiki/Main_Page

Table 1: Characteristics of Text Collections

Collection	Alphabet Size	Average LCP
DNA	16	59
English	239	9390

6.2 Improving Cache Performance

A set of experiments is presented below to demonstrate that the proposed cache-aware algorithms exhibit better cache performance and run faster than their cache-unaware counterparts. In all the experiments presented in this section, a single core was utilized to construct a suffix tree. The wall-clock time required to construct and store the suffix tree in memory was measured in all the experiments; the input string was preloaded in memory. Intel's VTune performance analyzer was utilized to measure performance counters such as TLB and L2 cache misses. We also report the memory usage of the proposed algorithms.

6.2.1 Evaluation of CMPUTree

Initially, we compare the performance of *CMPUTree* against that of Ukkonen's algorithm. We consider two variants of the *CMPUTree* algorithm, namely *CMPUTree_E* and *CMPUTree_PC*. *CMPUTree_PC* utilizes the *PreCache* partitioning algorithm (Section 4). In contrast, *CMPUTree_E* employes the variable length, prefix-based partitioning technique proposed in [22]. The latter is an exact partitioning technique that performs multiple passes over the input string. It has been utilized in several disk-based suffix tree construction algorithms to effectively partition skewed datasets such as DNA sequences. Since the suffix links recovery phase is optional, we also consider a variant that does not apply this phase, termed *nsl* (no suffix links).

In Figure 5, we present the suffix tree construction time of these algorithms as we increase the size of a DNA sequence from 20MB to 80MB. The memory usage of *CMPUTree* is reported in Table 2. For the entire range of string sizes examined, *CMPUTree_PC* performs $1.5X - 1.6X$ faster than Ukkonen's algorithm and $1.2X - 1.7X$ faster than *CMPUTree_E*. For small string sizes, the overhead of the partitioning technique is small, and *CMPUTree_E* performs better than Ukkonen's algorithm. However, as the string size becomes larger, the improved cache performance of the suffix tree construction phase does not compensate for the overhead introduced by the partitioning algorithm and *CMPUTree_E* performs worse than Ukkonen's algorithm (when the input string is 80MB in our experiment). In contrast, *CMPUTree_PC* consistently outperforms both *CMPUTree_E* and Ukkonen's algorithm, verifying its improved cache performance and the effectiveness of our low overhead partitioning algorithm.

The same experiment was conducted using English text data (Figure 6). As in the case of DNA sequences, *CMPUTree_PC* consistently performs $1.3X - 1.5X$ faster than both

Table 2: Memory Usage of *CMPUTree*

String Size (MB)	Memory Usage (GB)
20	2.1
40	4.3
60	6.7
80	8.9

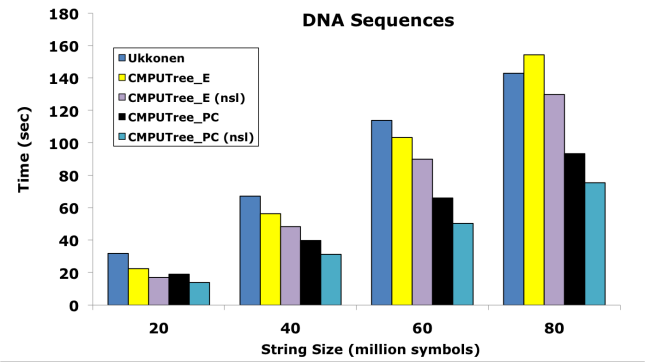

Figure 5: Evaluation of *CMPUTree* for biological text data.

Ukkonen's algorithm and *CMPUTree_E*. The performance difference between *CMPUTree* and Ukkonen's algorithm is smaller compared to the case of biological text data for a given input string size. Due to the larger average LCP of the English text collection, *CMPUTree* underestimates the space overhead of some edge labels, resulting in higher number of random memory accesses while traversing a partially constructed suffix tree.

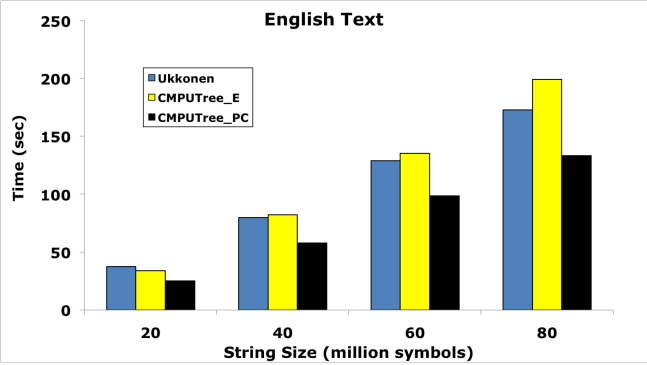

Figure 6: Evaluation of *CMPUTree* for English text data.

In the next experiment, the goal is to study the tradeoffs in using a sample-based partitioning technique. Figure 7 presents the execution time breakdown of *CMPUTree_E* and *CMPUTree_PC* algorithms on different phases when the input string size is 80MBs. The cost of the merging phase is not reported as it is negligible in this experiment. The sample-based partitioning technique clearly reduces the partitioning cost. However, since it relies on estimates of the number of suffixes of each partition, it typically produces more partitions (depending on the sample size) than the exact method (applied in *CMPUTree_E*). Although, the larger number of partitions increases the time spend on the remaining phases (construction, suffix link recovery, merging), the reduction of the partitioning cost compensates for the overhead introduced.

Table 3: Memory Usage of *MAPST*.

String Size (MB)	Memory Usage (GB)
20	0.8
40	1.7
60	2.7
80	3.6

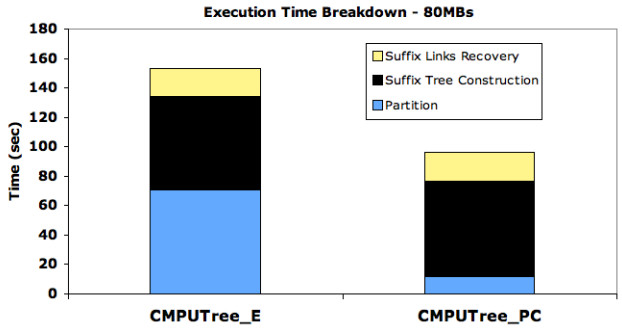

Figure 7: Execution time breakdown of *CMPUTree_E* and *CMPUTree_PC* algorithms when the input string size is 80MBs.

6.2.2 *Evaluation of MAPST*

The performance of the *MAPST* algorithm is assessed next. The suffix tree construction time of *MAPST* and *WOTD* is compared for biological and English text data. The results are presented in Figures 8 and 9, respectively. The memory usage of *MAPST* is reported in Table 3. For small string sizes (20MB in our experiment) *MAPST* and *WOTD* exhibit the same performance (Figure 8). However, as the string size increases from 20MB to 100MBs, their performance difference grows from 1% to 43%, respectively. The working set of the *WOTD* algorithm decreases as the construction of the suffix tree moves downwards, i.e. as lower nodes of the suffix tree are being evaluated. Hence, for the case of small strings, it takes only few node evaluations for the working set to fit in cache. As the string size increases, more tree nodes are evaluated by accessing memory resident data. Consequently, the performance difference between *WOTD* and *MAPST* is amplified, as the latter ensures that every node evaluation is performed in cache by utilizing *PreCache* and the compressed materialized prefixes.

For the case of English text data (Figure 9), the performance difference between *WOTD* and *MAPST* also escalates as we increase the string size, but at a lower pace compared to the case of biological data. For small strings (20MB in our experiment) *WOTD* is 20% faster than *MAPST* and when the string size is 100MBs, the latter is 27% faster. Compared to the case of biological data, the smaller performance difference is attributed to two reasons: a) the considerably larger alphabet size of English text, and b) the relatively higher overhead of LZW compared to RLE. The larger the alphabet size, the faster the reduction of the working set

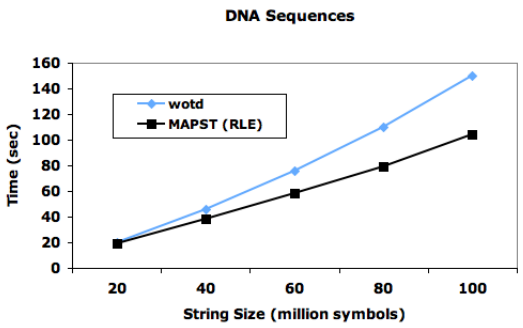

Figure 8: Evaluation of *MAPST* for biological text data.

size (factor of $|\Sigma|$) and consequently, fewer node evaluations are required for the working set of *WOTD* to fit in cache (for a given string size).

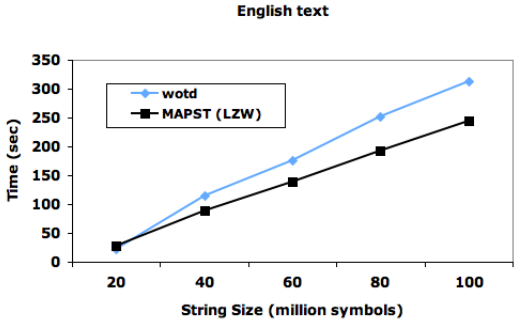

Figure 9: Evaluation of *MAPST* for English text data.

In the next experiment, we study the effect of prefix size on the performance of *MAPST*. Figure 10 presents the time required to construct a suffix tree from a DNA sequence of size 100MBs as we vary the number of RLE codes (k) of each materialized prefix. As we increase the prefix size to 8 codes, more nodes are evaluated from the materialized prefixes before they are exhausted, thus improving the cache performance of *MAPST*. However, as we increase the value of k, *MAPST* performs worse. Larger values of k increase the space requirements of each partition, the number of partitions created and the compression cost. The same effect was observed with English text data; the results are omitted due to space constraints.

In order to properly set the value of k, we must have explicit knowledge of the characteristics (distribution of the lcp) of a text collection. If that knowledge is available, we can compute the value of k that minimizes the expected number of memory accesses. If this is not the case, sampling could be utilized to provide this information.

6.2.3 *Under the Microscope*

In all the aforementioned experiments, the wall-clock time required to construct a suffix tree is measured and it is demonstrated that the proposed cache-conscious algorithms perform better than their cache-unaware counterparts. Next, we use hardware performance counters to corroborate that

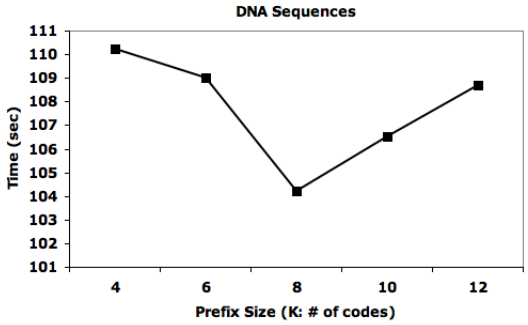

Figure 10: Effect of prefix size on execution time of *MAPST* for biological data.

this performance improvement is attributed to the reduction of cycles wasted on various stalls and to the improved CPU utilization.

In Figure 11, we present the normalized execution time breakdown (in cycles) of all the algorithms that were examined in our experimental evaluation for processing a DNA sequence of size 100MBs. It can be seen that all cache-aware algorithms increase the amount of time spent on useful computation and reduce the number of cycles wasted on stalls. *CMPUTree* reduced the L2 cache misses by 63% and the TLB misses by 80%. The wasted cycles accounted for the 82% of the total execution time for the case of Ukkonen's algorithm and for the 49% for the case of the *CMPUTree* algorithm. Significant reduction of the number of wasted cycles is also reported for the case of the *MAPST* algorithm.

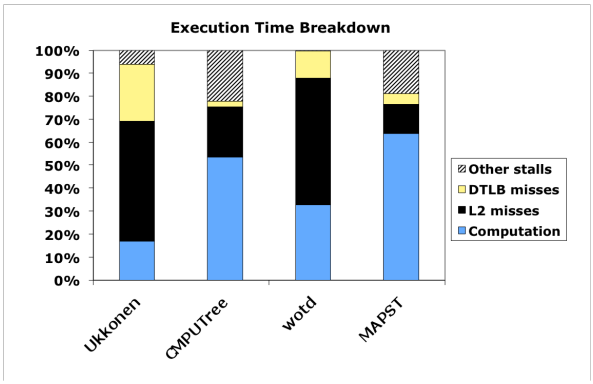

Figure 11: Execution time breakdown of suffix tree construction algorithms.

6.3 Exploiting on-chip Parallelism

In this section, we study the ability of the proposed algorithms to exploit the processing elements (cores) of a CMP. In Figures 12 and 13, we demonstrate the performance of *CMPUTree* and *MAPST*, respectively for processing a DNA sequence of size 100MBs as we increase the number of utilized cores. We present the normalized execution time with respect to the case when a single core is utilized.

As illustrated in Figures 12 and 13, the proposed algorithms effectively utilize the computational power of CMPs, significantly reducing the suffix tree construction cost. In both cases, the execution time is reduced by a factor of 6 when eight cores are utilized (in our experiment). The non-

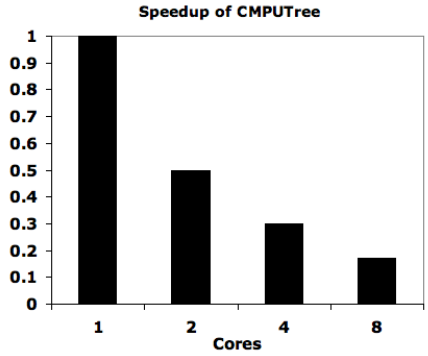

Figure 12: Speedup of CMPUTree as we increase the number of cores utilized.

linear speedup is attributed to the approximate partitioning technique and the online scheduling algorithm applied to assign the partitions to cores. This experiment demonstrates the effectiveness of the *PreCache* partitioning algorithm in parallelizing the suffix tree construction task. This experiment was also conducted with English text data with the same results, verifying that algorithms designed for CMP architectures can significantly reduce the suffix trees construction cost.

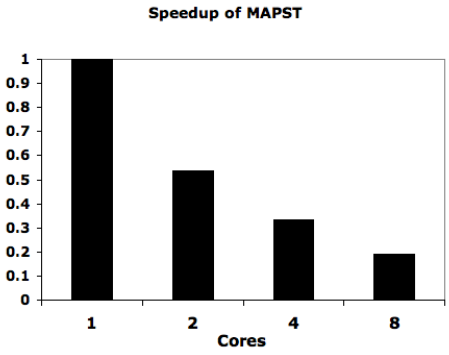

Figure 13: Speedup of MAPST as we increase the number of cores utilized.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we studied the problem of improving the performance of in-memory suffix tree construction algorithms. Initially, we proposed *PreCache*, a low-overhead cache partitioning algorithm that is utilized as a building block to improve the cache performance and parallelize the suffix tree construction task. We proposed, *CMPUTree* and *MAPST*, two novel suffix tree construction algorithms that are tailored to CMP architectures. Through a detailed experimental evaluation, we demonstrated that the algorithms proposed exhibit improved cache performance and effectively utilize the computational power of CMP architectures, thus achieving very good speedup.

8. REFERENCES

[1] A. Apostolico and W. Szpankowski. Self-alignment in words and their applications. *J. Algorithms*, 13:446–467, 1992.

[2] S. J. Bedathur and J. R. Haritsa. Engineering a fast online persistent suffix tree construction. In *ICDE*, page 720, 2004.

[3] P. Bieganski. *Genetic sequence data retrieval and manipulation based on generalized suffix trees*. PhD thesis, University of Minnesota, 1995.

[4] A. M. Carvalho, A. L. Oliveira, A. T. Freitas, and M.-F. Sagot. A parallel algorithm for the extraction of structured motifs. In *SAC*, pages 147–153, 2004.

[5] C. Chen and B. Schmidt. Constructing large suffix trees on a computational grid. *Journal of Parallel and Distributed Computing*, 66(12):1512–1523, 2006.

[6] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Inspector joins. In *VLDB*, pages 817–828, 2005.

[7] C.-F. Cheung, J. X. Yu, and H. Lu. Constructing suffix tree for gigabyte sequences with megabyte memory. *IEEE TKDE*, 17(1):90–105, 2005.

[8] J. Cieslewicz and K. A. Ross. Adaptive aggregation on chip multiprocessors. In *VLDB*, pages 339–350, 2007.

[9] I. Coorporation. Intel 64 and IA-32 architectures optimization reference manual, May 2009.

[10] M. Farach-Colton, P. Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *J. ACM*, 47(6):987–1011, 2000.

[11] B. Gedik, R. R. Bordawekar, and P. S. Yu. Cellsort: high performance sorting on the cell processor. In *VLDB*, pages 1286–1297, 2007.

[12] A. Ghoting and K. Makarychev. Serial and parallel methods for I/O efficient suffix tree construction. In *SIGMOD '09*, pages 827–840, 2009.

[13] R. Giegerich, S. Kurtz, and J. Stoye. Efficient implementation of lazy suffix trees. *Software - Practice and Experience*, 33:1035–1049, 2003.

[14] D. Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge University Press, 1997.

[15] R. Hariharan. Optimal parallel suffix tree construction. In *STOC*, pages 290–299, 1994.

[16] E. Hunt, M. P. Atkinson, and R. W. Irving. A database index to large biological sequences. In *VLDB*, pages 139–148, 2001.

[17] J. Kärkkäinen and E. Ukkonen. Sparse suffix trees. In *COCOON*, pages 219–230, 1996.

[18] A. Konig, K. Church, and M. Markov. A data structure for sponsored search. In *ICDE*, pages 90–101, 2009.

[19] S. Kurtz. Reducing the space requirement of suffix trees. *Softw. Pract. Exper.*, 29(13):1149–1171, 1999.

[20] G. Landau, B. Schiever, and U. Vishkin. Parallel construction of a suffix tree. *Lecture Notes in Computer Science*, 267:314–325, 1987.

[21] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976.

[22] B. Phoophakdee and M. J. Zaki. Genome-scale disk-based suffix tree indexing. In *SIGMOD*, pages 833–844, 2007.

[23] J. Rao and K. A. Ross. Making B+- trees cache conscious in main memory. *SIGMOD Rec.*, 29(2):475–486, 2000.

[24] B. Schieber and U. Vishkin. On finding lowest common ancestors: simplification and parallelization (extended summary). pages 111–123, 1988.

[25] A. Shatdal, C. Kant, and J. F. Naughton. Cache conscious algorithms for relational query processing. In *VLDB*, pages 510–521, 1994.

[26] Y. Tian, S. Tata, R. A. Hankins, and J. M. Patel. Practical methods for constructing suffix trees. *The VLDB Journal*, 14(3):281–299, 2005.

[27] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[28] P. Weiner. Linear pattern matching algorithms. In *In Proceedings of the 14th Annual Symposium on Switching and Automata Theory*, IEEE, 1973.

[29] M. Yue. A simple proof of the inequality ffd(l) < (11/9)opt(l) + 1, for all l, for the ffd bin-packing algorithm. *Acta Mathematicae Applicatae Sinica*, 7:321–331, 1991.