A parallel pattern-growth algorithm

Parallel pattern-growth

Kenmogne Edith Belise*,a — Nkambou Roger** — Tadmon Calvin* — Engelbert Mephu Nguifo***

* Department of Mathematics and Computer Science, LIFA
University of Dschang, Cameroon
ebkenmogne@gmail.com

** Computer Science Department, Knowledge Management laboratory
University of Québec at Montréal, Canada

*** Computer Science Institute - LIMOS - UMR CNRS 6158
Complexe scientifique des cézeaux, 1 rue de la chebarde, 63178 Aubière cedex, France

RSUM. La recherche des motifs séquentiels est un problème important en fouille de données largement abordée par la communauté de fouille de données, avec un très grand champ d'applications. La recherche des motifs séquentiels vise à extraire un ensemble d'attributs d'un nombre important d'objets collectés dans une base de données. De ce fait, les algorithmes d'extraction des motifs séquentiels sont bien connus pour la consommation à la fois du temps et de la mémoire pour de grandes bases de données. De plus, de nombreuses applications sont critiques en termes de temps d'excécution et impliquent d'énormes volumes de données. De telles applications exigent une puissance d'extraction que les algorithmes séquentiels ne peuvent fournir. Ainsi, il est clairement important d'étudier des algorithmes parallèles. Le travail présenté dans ce papier est orienté vers la conception d'une version parallèle de *prefixSuffixSpan* pour les architectures multi-coeurs en utilisant la méthode de parallélisation PCAM. Nous avons testé notre algorithme parallèle en utilisant plusieurs ensembles de données réelles. Nos expériences ont montré des performances interessantes en termes de vitesse et d"accélération pour presque tous les cas.

ABSTRACT. Sequential pattern mining is an important data mining problem widely addressed by the data mining community, with a very large field of applications. The sequence pattern mining aims at extracting a set of attributes, shared across time among a large number of objects in a given database.Thereby, sequential pattern mining algorithms are well known to be both time and memory consuming for large databases. Moreover many applications are time-critical and involve huge volumes of data. Such applications demand more mining power than serial algorithms can provide. Thus, it is clearly important to study parallel sequential-pattern mining algorithms that take advantage of the computation. The work presented in this paper is directed towards the design of a parallel version of *prefixSuffixSpan* for multi-core architectures using the PCAM parallelization method. We have tested our algorithm using several real-life data sets. Our experiments showed good speedups and accelerations for almost all the cases.

MOTS-CLS : croissance-de-motifs, algorithme parallèle, découverte des motifs séquentiels, vitesse, acceleration

KEYWORDS : pattern-growth, parallel algorithm, sequential pattern discovery, speedup, acceleration

1. Introduction

Sequential pattern mining is a challenging problem since the mining may have to generate or examine a combinatorially explosive number of intermediate subsequences. Thereby, sequential pattern mining algorithms are well known to be both time and memory consuming for large databases. To make sequential pattern mining practical for large data sets, the mining process must be efficient, scalable, and have a short response time. Moreover, since sequential pattern mining requires iterative scans of the sequence dataset with numerous data comparison and analysis operations, it is computationally intensive. Furthermore, many applications are time-critical and involve huge volumes of data. Such applications demand more mining power than serial algorithms can provide. Thus, it is clearly important to study parallel sequential-pattern mining algorithms that take advantage of the computation. Although a significant amount of research results have been reported on serial implementations [18, 9, 8, 6, 13, 22, 3, 14, 16, 17, 7] of sequential pattern mining, there is still much room for improvement in its parallel implementation [20, 21, 15].

The best algorithms for both frequent itemset mining problem and sequential pattern mining problem are based on pattern-growth, a divide-and-conquer algorithm that projects and partitions databases based on the currently identified frequent patterns and grow such patterns to longer ones using the projected databases. We have proven in paper [12, 11] that our sequential pattern-growth algorithm, baptised *prefixSuffixSpan*, outperforms the best previously known sequential pattern-growth algorithm, called *PrefixSpan*. In this paper, we design a parallel version of *prefixSuffixSpan* for multi-core architectures.

The sequel of this paper is organized as follows. Section 2 presents the PCAM parallelization method. Section 3 presents new results. Sub-section 3.1 studies the parallelization of *prefixSuffixSpan*. Sub-section 3.2 designs a multi-core version of the *prefixSuffixSpan* algorithm. Sub-section 3.3 is devoted to the implementation of the multi-core version of *prefixSuffixSpan* and performance analysis. The experimental results show that our parallel algorithm usually achieve interesting speedups. Concluding remarks are stated in section 4.

2. The PCAM parallelization method

In this section, we present the PCAM parallelization method [4]. PCAM stands for Partitioning, Communication, Agglomeration and Mapping. This method organizes the design of a parallel algorithm from a sequential algorithm into four steps. The starting step dealts with the partitioning of the overall computations into tasks. The second step dealts with communications among tasks. The third step studies possible agglomerations of tasks in order to obtain bigger tasks. The fourth step dealts with the mapping, also called allocation, of tasks onto available processors.

The partitioning [19, 1, 2] decomposes the overall computations into either *fine-grain*, *medium-grain* (also called *coarse-grain*) or *large-grain* tasks, depending on the granularity, i.e. size in term of computations, of tasks. A *fine-grain* task [1] consists of a constant number basic operations. A *medium-grain* task [5] consists of a linear number of basic operations. In many sequential algorithms, it is on the form of a depth-one loop whose body computes a constant number of basic operations. A *large-grain* task [5] consists of a large number of basic operations. In many sequential algorithms, it is on the form

of a loop of depth greater than one whose body computes a constant number of basic operations.

The study of communications involves the identification of data to be transferred between tasks as well as the definition of related data structures and of reliable communication protocols for data exchanges beetween tasks. A classic challenging problem is to design communication protocols that optimize communication costs [2]. A non-adequate communication protocol may significantly slows down the execution of the corresponding parallel algorithm. Because of this, the commnication protocol should fit with the allocation of tasks to processors.

The study of agglomerations leads to a medium-grain decomposition from a fine-grain decomposition and to a large-grain decomposition from a medium-grain decomposition. Although agglomerations of large-grain tasks lead to bigger tasks, the granularity of the new decomposition obtained remains unchanged. By gathering tasks, the number of data transfers between them are reduced. Thus, agglomerations contribute significantly to the optimization of communication costs [5]

The mapping consists in assigning tasks obtained from agglomerations to processors so as to minimize communications costs and the sum of idles times of all the procesors used in the parallel algorithm [1, 2, 5]

3. New results

3.1. Parallelizing prefixSuffixSpan

3.1.1. Partitioning prefixSuffixSpan and studying communications therein

In this section, we design a multi-core version of prefixSuffixSpan. This is done following the PCAM parallelization method [4]. At the first glance, prefixSuffixSpan can be decomposed into *projection tasks*. The unique level-one *projection task* takes as input the global dataset and a pattern-growth direction [10], mines frequent items and generates one level-one projected dataset per frequent item. Each non-empty level-one dataset leads to a level-two projection task which takes as input a frequent item, a level-one projected dataset and a pattern-growth direction, and generates length-two sequential patterns and one level-two projected dataset per length-two pattern generated. Each non-empty level-two dataset, in turn, leads to a level-three projection task which takes as input a length-two sequential pattern of the form $\alpha.\alpha\prime$, a level-two projected dataset and a pattern-growth direction, and generates length-three sequential patterns by making grow either prefix α or suffix $\alpha\prime$ and one level-three projected dataset per length-three pattern generated.

More generaly, by considering that the global dataset is of level zero , a level-k projection task takes as input (1) a length-k sequential pattern of the form $\alpha.\alpha\prime$, (2) a level-(k-1) projection dataset, and (3) a pattern-growth direction. If the pattern-growth direction is *left-to-right* (resp. *right-to-left*) it makes grow prefix α (resp. suffix $\alpha\prime$). It generates length-(k+1) sequential patterns and one level-(k+1) projected dataset per generated pattern. In this partitioning, the only task to be executed at the beginning is the level-one task. Because of this, only one thread can work at the beginning while the others threads are waiting for the end of the execution of the level-one task. Thus, this first partitioning is not suitable in minimizing idle times of threads involved in the parallel execution of prefixSuffixSpan. As a consequence, it can be improved.

3.1.2. Partitioning the level-one task and studying communications and synchronization therein

The level-one projection task should be partitioned into a number of parallel smaller tasks, i.e. tasks that could be executed simultaneously, in order to allow all thread to get a task to execute at the beginning. This is done here in eight steps following partitioning techniques developped in [1, 2]. The number of tasks of each step from step 2 to step 7 is equal to the number of threads involved in the parallel execution of prefixSuffixSpan. These steps are described here as follows :

1) *Step 1 :* The global dataset is partitioned into as many partial data sets as there are threads devoted to the parallel execution of prefixSuffixSpan.

2) *Step 2 :* Each thread gets a partial dataset and computes the partial supports of items therein in order to obtain partial supports.

3) *Step 3 :* Partial supports are used to update global supports The update is done by the thread who has computed the partial supports. Partial supports should be stored in a concurrent data structure because of concurrent write operations involving global supports and arisen from many threads. A synchronization barrier is needed here because the next step should begin after the end of this one. It can be done by using a concurrent integer data to count the number of threads who have update the global supports. Such an integer is initialized to zero and incremented after each update of global supports.

4) *Step 4 :* Each thread gets the global supports per item and a partial list of items, then seeks for frequent items in its list of items in order to obtain a partial list of frequent items.

5) *Step 5 :* Partial lists of frequent items are used to update the global list of frequent items. The update is done by the thread who has constructed the partial list. Partial lists should be stored in a concurrent data structure because of concurrent write operations involving the global list and arisen from many threads. A synchronization barrier is needed here because the next step should begin after the end of this one. It can be done by using a concurrent integer data to count the number of threads who have update the global list of frequent items. Such an integer is initialized to zero and incremented after each update of the global list of frequent items.

6) *Step 6 :* Each thread gets the global list of frequent items and computes the left and right weights of the sequences of its dataset received at step 2.

7) *Step 7 :* Partial left (resp. right) weights are used to update the global left (resp. right) weight assuming that it is initialized to zero. The global left and right weights are used to determine the promising pattern-growth direction. The update is done by the thread who has calculated the partial weights. Synchronization issues arisen here are solved as in steps 3 and 5.

8) *Step 8 :* Tasks of this step represent the new level-one projection tasks. Each frequent item leads to such a task.

3.1.3. An improved Partitioning of prefixSuffixSpan

The main weakness of the first partitioning is overcome here by replacing the level-one task with its decomposition into smaller (in term of the amount of computations) tasks. A new partitioning is obtained by replacing level-one task with its decomposition. This leads to an improved partitioning of *prefixSuffixSpan*. It reduces the idle times of threads compared to the previous partitioning.

3.1.4. Issues related to the improved partitioning

Agglomerations : We use an integer value called *depth* which indicates the projection-task level from which agglomerations should be constructed. If the depth value is d, agglomerations are constructed only from the projection tasks of level greater than $d-1$. An agglomeration is obtained by gathering a level-d task with all its descendents. As a consequence, once a thread retrieves a level-d projection task from the pool of projection tasks, it executes that task with all its descendents. The resulting partitioning is a mixture of medium-grain and large-grain tasks. Large-grain tasks permit to reduce the synchronization costs arisen from the handling of the pool of projection tasks.

The concurrent pool of projection tasks : A concurrent pool of tasks is used to handle the storage and retrieval of projection tasks. Once a thread generates a projection task of level lower than the value of *depth*, it saves that task in the pool if the pool is not full. Otherwise, it should execute that generated projection task. Idle threads retrieve projection tasks to execute from the pool. This pool reduces idle times of threads by providing tasks to idle threads.

Mapping : The mapping of tasks onto threads is unknown before the beginning of the parallel execution of *prefixSuffixSpan*. Tasks are assigned to threads during the parallel execution. Because of this, the mapping is dynamic. As mentioned above, idle threads retrieve tasks to execute from the concurrent pool of projection tasks. This contributes to load balancing calculations.

Communications : Communications between threads are performed through four concurrent data structures. As each data structure is a critical resource, it can not be used by two threads simultaneously. The costs [5] of the handling of synchronization related to a concurrent data structure increases with the number of threads needing to access that data structure. This may cause a slow down of the acceleration of the parallel algorithm when the number of threads increases.

Termination criterion of the multi-core algorithm : A concurrent array called *busy* is used. Cell $busy[i]$ contains 1 if the thread numbered i has gotten a projection task from the concurrent pool of projection tasks during its last attempt and 0 otherwise. If all the cell of array *busy* contain 0, it means that no thread has a projection task to execute. When this condition is satisfied, the multi-core algorithm ends.

3.2. A multi-core version prefixSuffixSpan

In this section, we translate the results of section 3.1 into a multi-core version of *prefixSuffixSpan*. Here is the list of functions executed by all thread involved in the multi-core execution of *prefixSuffixSpan*.

1) Function THREADTASKFORSUPPORTCOUNT is a translation of steps 2 and 3 into an algorithm. It is executed by a thread to (1) compute the partial supports per item of its partial dataset, (2) update the global supports per item with its partial supports, (3) wait for all the updates of global supports, and (4) get the global supports.

2) Function THREADTASKTOFINDFREQUENTITEM is a translation of steps 4 and 5 into an algorithm. It is executed by a thread to (1) construct its partial list of frequent items from its partial list of items, (2) update the global list of frequent items with its

partial list of frequent items, (3) wait for all the updates of global list of frequent items, and (4) get the global list of frequent items.

3) Function THREADTASKTOGETGROWTHDIRECTION is a translation into an algorithm. It is executed by a thread to (1) compute its partial left and right weights of its partial dataset, (2) update the global left and right weights with its partial left and right weights, (3) wait for all the updates of global left and right weights, (4) get the global weights, and (5) determine the pattern-growth direction from global weights.

4) Function PROJECTIONTASK is a translation of the description of projection tasks into an algorithm. It is used by a thread to execute a projection task.

5) Function MAINTHREADTASK is the starting point of the execution of all thread involved in the multi-core execution of *prefixSuffixSpan*. The others functions are called in this one.

3.3. Implementation and performance analysis

The data sets used here are collected from the webpage (http ://www.philippe-fournier-viger. com/spmf/index.php) of SPMF software. This webpage provides large data sets in SPMF format that are often used in the data mining litterature for evaluating and comparing algorithm performance. All experiments are done on a 32-cores. All the algorithms are implemented in Java and grounded on SPMF software [17]. The experiments consisted of running the multi-core version of *prefixSuffixSpan* on each data set and for a given number of threads ranging from two to thirty two while decreasing the support threshold until algorithms became too long to execute or ran out of memory. We also studied the influence of the depth's value on the algorithm's performance when the number of threads is thirty two. For each execution, we recorded the execution times, the speed up and the accelerations. The speed up of a parallel execution is defined as follows.

$$\text{Speed up for n threads} = \frac{\text{execution time for one thread}}{\text{execution time for n threads}}$$

The acceleartion of a parallel execution is defined as follows.

$$\text{Acceleration for n threads} = \frac{\text{speed up for n threads}}{\text{n}}$$

The speed up is upperly bounded by the number of threads while the acceleration is upperly bounded by 1. In the following, we analyze the performance of our multi-core algorithm per data set. The experimentations show that the speed up may increase (1) as the number of threads increases, (2) as the depth increases, (3) as the support threshold decreases, and (4) as the number of sequential patterns increases. They also show that the speed up may be very sensitive to the change of depth. The acceleration of our parallel algorithm on four real-life data sets is within range $[0.58\ 1]$ for minimum support thresholds and thirty two threads. In [20], a parallel version of the well known *PrefixSpan* algorithm is proposed. The acceleration of that parallel algorithm on five synthetic data sets is within range $[0.25\ 0.5]$ [20] for minimum support thresholds and thirty two processorss. However, a divide-and-conquer property, though minimizing inter-processor communication, causes load balancing problems, which restricts the scalability of parallelization.In [20], synthetic data sets show better speed up than real ones. This is because synthetic data sets have more frequent items and, after the large projected databases are partitioned, the sub-databases derived are of similar size. However, in real data sets, the number of frequent items is small and even when the large tasks are partitioned into smaller subtasks, the size

of the subtasks may still be larger, or even much larger. It is clear that the performance of our parallel algorithm is better compared to the performance of the parallel version of PrefixSpan proposed in [20].

4. Conclusion

In this paper, we have proposed a parallel implementation of the *prefixSuffixSpan* mining algorithm. This parallel version of *prefixSuffixSpan* is obtained in four main steps : (1) partitioning of *prefixSuffixSpan* into tasks following the PCAM parallelization method, (2) studying issues related to the partitioning, namely (2.1) agglomerations, (2.2) the concurrent pool of projection tasks, (2.3) mapping, (2.4) communications and synchronization, and (2.5) the termination criterion, (3) translating tasks into algorithms, and (4) implementing algorithms.

We have tested our algorithm using several real-life data sets. Our experiments showed good speedups and accelerations for almost all the cases. These results outperform the best previous ones [20].

5. Bibliographie

[1] CLÉMENTIN TAYOU DJAMEGNI , PATRICE QUINTON , SANJAY V. RAJOPADHYE , TANGUY RISSET , MAURICE TCHUENTE, « A reindexing based approach towards mapping of DAG with affine schedules onto parallel embedded systems », *J. Parallel Distrib. Comput.*, vol. 69, n° 1, 1–11, 2009.

[2] CLÉMENTIN TAYOU DJAMEGNI , MAURICE TCHUENTE, « A Cost-Optimal Pipeline Algorithm for Permutation Generation in Lexicographic Order », *J. Parallel Distrib. Comput.*, vol. 44, n° 2, 153–159, 1997.

[3] CHIA-YING HSIEH , DON-LIN YANG , JUNGPIN WU, « An Efficient Sequential Pattern Mining Algorithm Based on the 2-Sequence Matrix », *Workshops Proceedings of the 8th IEEE International Conference on Data Mining (ICDM 2008), December 15-19, 2008, Pisa, Italy*, 583–591, 2008.

[4] IAN T. FOSTER, « Designing and building parallel programs - concepts and tools for parallel software engineering », *Addison-Wesley*, 1995.

[5] JEAN FRANÇOIS DJOUFAK KENGUE , PETKO VALTCHEV , CLÉMENTIN TAYOU DJAMEGNI, « Parallel Computation of Closed Itemsets and Implication Rule Bases », *Parallel and Distributed Processing and Applications, 5th International Symposium, ISPA 2007, Niagara Falls, Canada, August 29-31, 2007, Proceedings*, 359–370, 2007.

[6] JIAN PEI , JIAWEI HAN , BEHZAD MORTAZAVI-ASL , JIANYONG WANG , HELEN PINTO , QIMING CHEN , UMESHWAR DAYAL , MEICHUN HSU, « Mining Sequential Patterns by Pattern-Growth : The PrefixSpan Approach », *IEEE Trans. Knowl. Data Eng.*, vol. 16, n° 11, 1424–1440, 2004.

[7] JIAWEI HAN , MICHELINE KAMBER, « Data Mining : Concepts and Techniques », *Morgan Kaufmann*, 2000.

[8] JIAWEI HAN , JIAN PEI , YIWEN YIN, « Mining Frequent Patterns without Candidate Generation », *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA.*, 1–12, 2000.

[9] KARAM GOUDA , MOSAB HASSAAN , MOHAMMED J. ZAKI, « Prism : An effective approach for frequent sequence mining via prime-block encoding », *J. Comput. Syst. Sci.*, vol. 276, n°

1, 88–102 2010.

[10] KENMOGNE EDITH BELISE, « The Impact of the Pattern-Growth Ordering on the Performances of Pattern Growth-Based Sequential Pattern Mining Algorithms », *Computer and Information Science*, vol. 10, n° 1, 23–33 2017.

[11] KENMOGNE EDITH BELISE , TADMON CALVIN , ROGER NKAMBOU, « A pattern growth-based sequential pattern mining algorithm called prefixSuffixSpan », *EAI Endorsed Trans. Scalable Information Systemsurnal*, vol. 4, n° 12, e4, 2017.

[12] KENMOGNE EDITH BELISE, « Contribution to the sequential and parallel discovery of sequential patterns with an application to the design of e-learning recommenders », *PhD Thesis. The University of Dschang, Faculty of Sciences, Department of Mathematics and Computer Science*, waiting for defense.

[13] LIONEL SAVARY , KARINE ZEITOUNI, « Indexed Bit Map (IBM) for Mining Frequent Sequences », *Knowledge Discovery in Databases : PKDD 2005, 9th European Conference on Principles and Practice of Knowledge Discovery in Databases, Porto, Portugal, October 3-7, 2005, Proceedings*, 659–666, 2005.

[14] MOHAMMED JAVEED ZAKI, « TSPADE : An Efficient Algorithm for Mining Frequent Sequences », *Machine Learning*, vol. 42, n° 1/2, 31–60 2001.

[15] MOHAMMED JAVEED ZAKI, « Parallel Sequence Mining on Shared-Memory Machines, *J. Parallel Distrib. Comput.*, vol. 61, n° 3, 401–426 2001.

[16] NIZAR R. MABROUKEH , CHRISTIE I. EZEIFE, « A taxonomy of sequential pattern mining algorithms », *ACM Comput. Surv.*, vol. 43, n° 1, 3 2010.

[17] PHILIPPE FOURNIER-VIGER , ANTONIO GOMARIZ , TED GUENICHE , AZADEH SOLTANI , CHENG-WEI WU , VINCENT S. TSENG, « SPMF : a Java open-source pattern mining library », *Journal of Machine Learning Research*, vol. 15, n° 1, 3389–3393 2014.

[18] RAKESH AGRAWAL , RAMAKRISHNAN SRIKANT, « Proceedings of the Eleventh International Conference on Data Engineering, March 6-10, 1995, Taipei, Taiwan », *Mining Sequential Patterns*, 3–14, 1995.

[19] SABEUR ARIDHI , LAURENT D'ORAZIO , MONDHER MADDOURI , ENGELBERT MEPHU NGUIFO, « Density-based data partitioning strategy to approximate large-scale subgraph mining », *Inf. Syst.*, vol. 48, 213–223 2015.

[20] SHENGNAN CONG , JIAWEI HAN , JAY HOEFLINGER , DAVID A. PADUA, « A sampling-based framework for parallel data mining », *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2005, June 15-17, 2005, Chicago, IL, USA*, 255–265, 2005.

[21] VALERIE GURALNIK , GEORGE KARYPIS, « Parallel tree-projection-based sequence mining algorithms », *Parallel Computing*, vol. 30, n° 4, 443–472 2004.

[22] ZHENGLU YANG , YITONG WANG , MASARU KITSUREGAWA, « LAPIN : Effective Sequential Pattern Mining Algorithms by Last Position Induction for Dense Databases », *Advances in Databases : Concepts, Systems and Applications, 12th International Conference on Database Systems for Advanced Applications, DASFAA 2007, Bangkok, Thailand, April 9-12, 2007, Proceedings*, 1020–1023 2007.

6. Annex

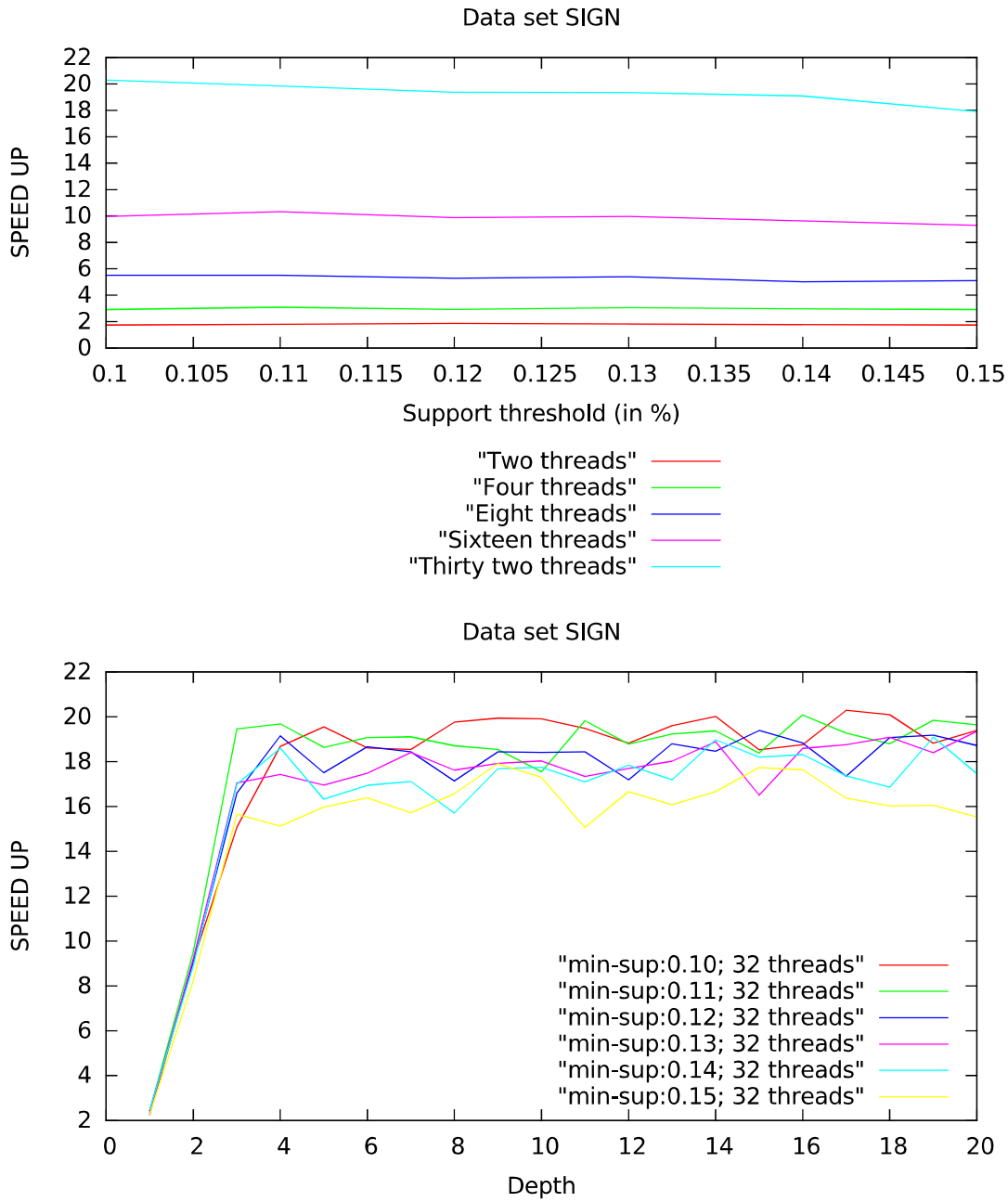

Figure 1. *Performances of the multi-core version of prefixSuffixSpan on the real-life data set SIGN. The speed up increases (1) as the number of threads increases, (2) as the depth increases in general up to* 4*, (3) slightly as the support threshold decreases. In the first part of the figure, the speed up is quite stable and the acceleration for the minimum support threshold is within range* $[0.63\ 0.86]$*.*

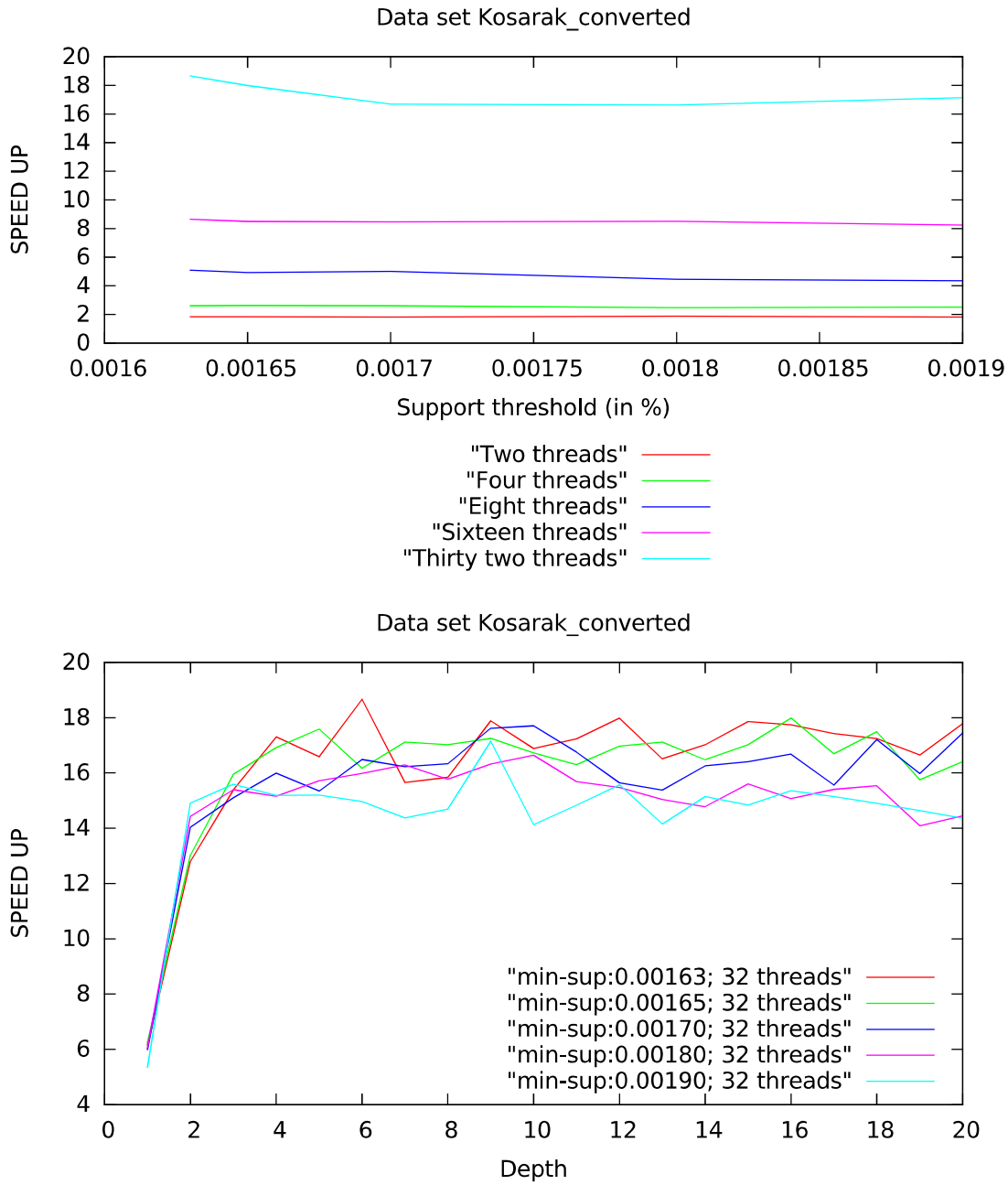

Figure 2. *Performances of the multi-core version of prefixSuffixSpan on the real-life data set Kosarak_converted. The speed up increases (1) as the number of threads increases, (2) as the depth increases in general, (3) as the support threshold decreases in general for thirty two threads. In the first part of the figure, the speed up is quite stable and the acceleration for the minimum support threshold is within range* $[0.58\ 0.91]$.

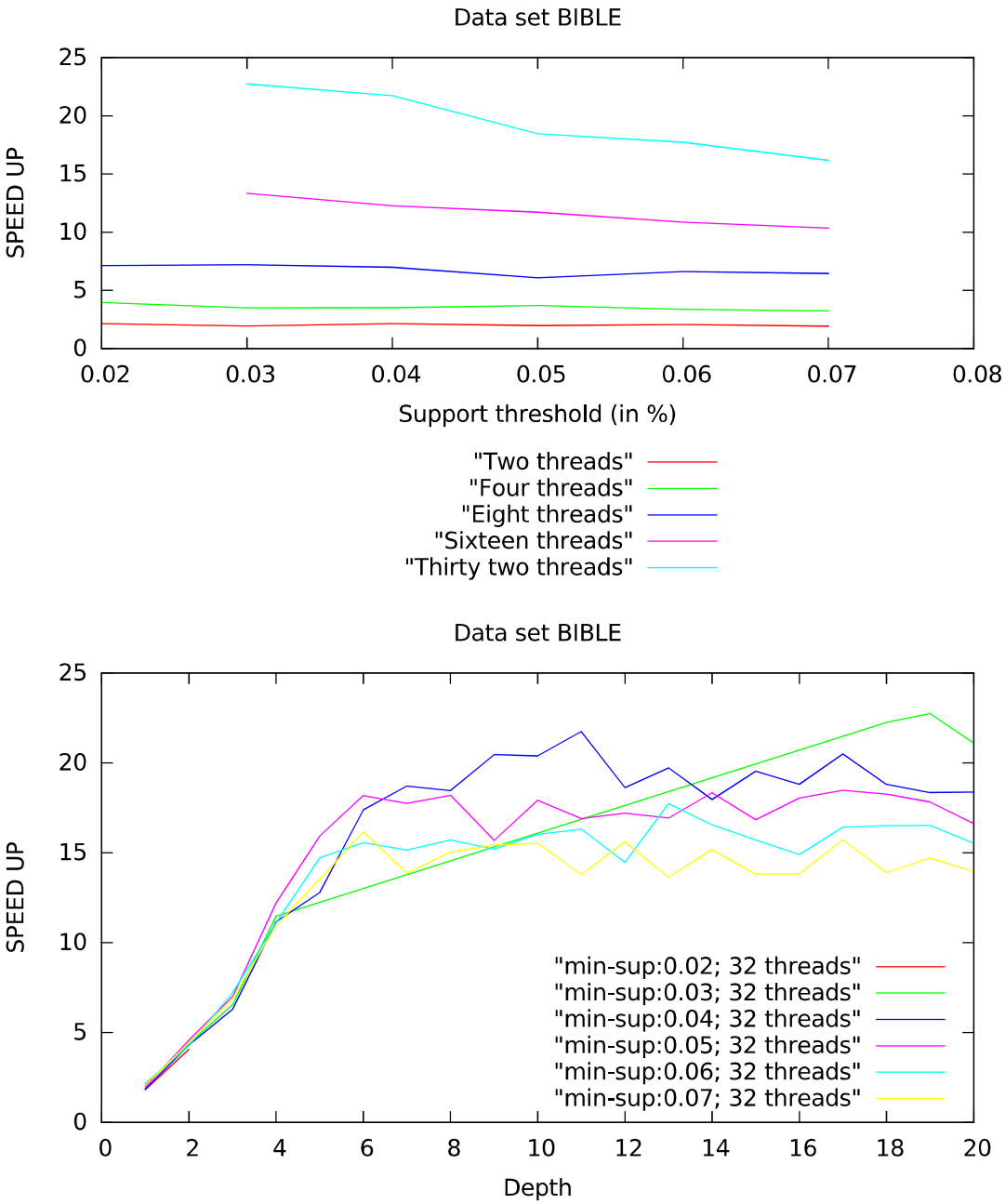

Figure 3. *Performances of the multi-core version of prefixSuffixSpan on the real-life data set BIBLE. The speed up increases (1) as the number of threads increases, (2) as the depth increases in general, (3) slightly as the support threshold decreases. In the first part of the figure, the speed up is relatively stable and the acceleration for the minimum support threshold is within range* $[0.71\ 1]$.

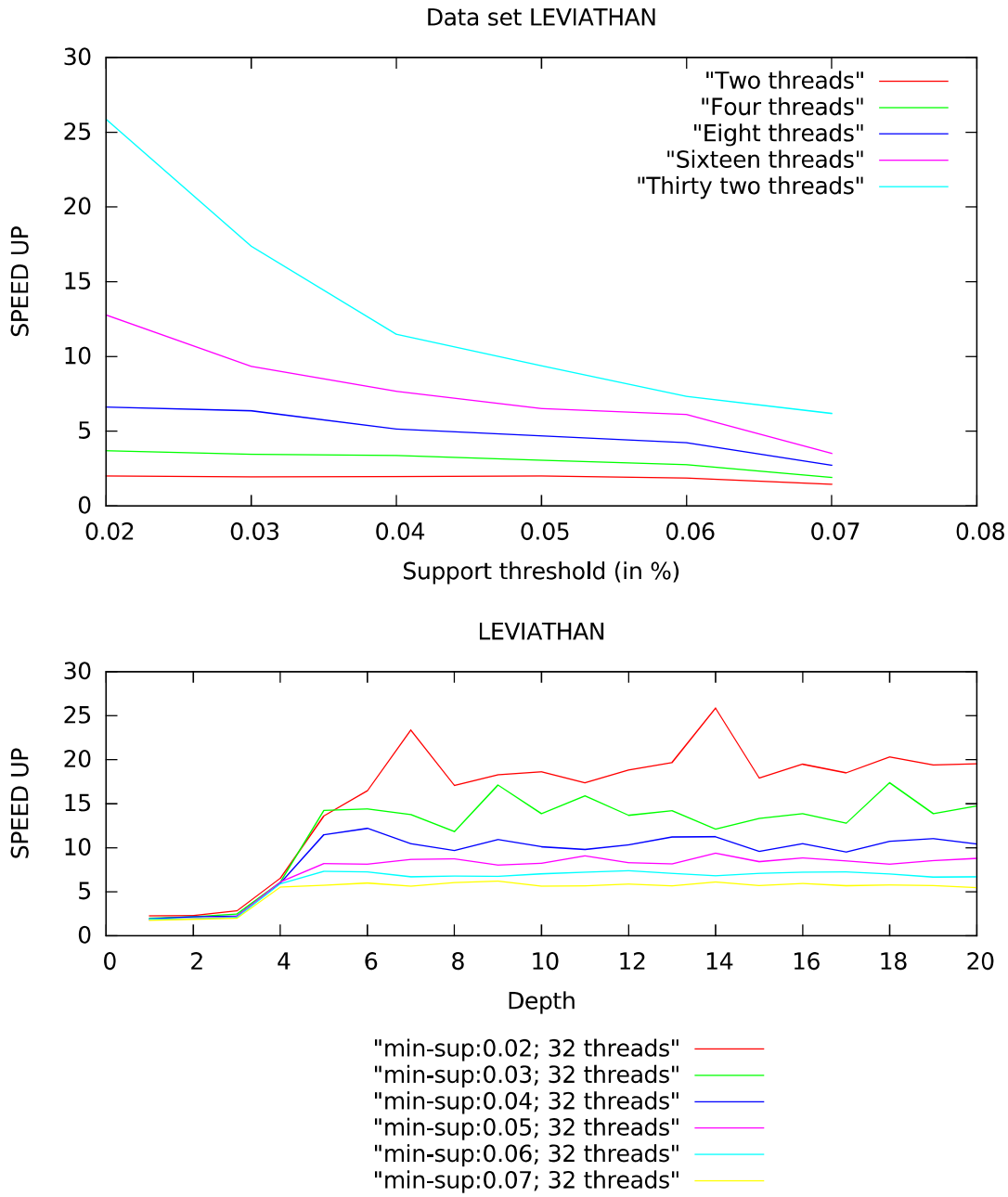

Figure 4. *Performances of the multi-core version of prefixSuffixSpan on the real-life data set LEVIATHAN. The speed up increases (1) as the number of threads increases, (2) as the depth increases in general up to 6, (3) as the support threshold decreases. In the first part of the figure, the speed up decreases significantly as the support threshold decreases when the number of threads is sixteen or twenty two, and the acceleration for the minimum support threshold is within range* $[0.80\ 1]$.