Research Article

International Journal of Advanced Computer Research, Vol 9(41)
ISSN (Print): 2249-7277 ISSN (Online): 2277-7970
http://dx.doi.org/10.19101/2015.16.17.7435

An efficient parallel framework for process discovery using OpenMP

Muktikanta Sahu[1*] and Gopal Krishna Nayak[2]
Assistant Professor, Department of Computer Science & Engineering, International Institute of Information Technology, Bhubaneswar, Odisha, India[1]
Professor, Department of Computer Science & Engineering, International Institute of Information Technology, Bhubaneswar, Odisha, India[2]

Received: 07-December-2018; Revised: 28-January-2019; Accepted: 31-January-2019
©2019 Muktikanta Sahu and Gopal Krishna Nayak. This is an open access article distributed under the Creative Commons Attribution (CC BY) License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Abstract
A process model is a graphical representation of the actual business process that is being executed. To build a process model from an event log, process discovery algorithms are used which are complex in nature and require prolonged execution as they involve extraction of the various ordering relations that exist between the events present in that event log. Given the exponential increase of data in event log, it is significant to have a robust and effective implementation of the computation intensive process discovery algorithms through parallel computing to generate a process model. Motivated by this theme the present work proposes a parallel computing approach to implement the Alpha algorithm for process discovery using the OpenMP application programming interface (API). An appropriate parallel programming framework to reduce the execution time by exploiting parallelism at the level of data, as well as task through a thorough analysis of the steps involved in the Alpha algorithm, has been developed. The effectiveness of the developed approach is presented on the basis of speedup factor through several experiments. The highest and the lowest speedups achieved were 13.24x and 4.71x respectively.

Keywords
Process model discovery, Alpha algorithm, OpenMP, Speedup.

1.Introduction
In the recent decade, process mining [1−6] has evolved as a novel discipline in data science. The basis of process mining combines features from the domains of data mining and business process intelligence. A comprehensive set of tools is available in the domain of process mining which can be used for process analysis and enhancement of existing business processes. As a result, many researchers have shifted their focus to process mining and its related areas to get better insights into business processes. The primary step in process mining is to discover process models and represent them in graphical form which is purely based on the information extracted from the recorded event log [1]. The other key areas of process mining are processed conformance checking and improvement of existing processes [1]. Thus, considering process discovery as the preliminary step in process mining, event log play an important role as they capture execution histories (or traces) of business processes.

The least three essential features that must be recorded in an event log on which process discovery techniques rely are: (i) name of the activity that has executed, (ii) a case identifier to which that particular activity belongs, and (iii) the completion timestamp of that activity [7]. Aggregating all the recorded events having the above three essential features constitute an event log. *Table 1* displays a sample event log consisting of 15 events in 5 cases. Process discovery algorithms try to correlate different events based on their respective case identifiers to form individual clusters that can subsequently be used to build a process model. The example event log given in *Table 1* contains three process instances from the case <*A, B, C*> and two process instances from the case <*A, D, E*>. The respective case identifiers for the recorded traces are 1, 2, 3, 4, and 5. The corresponding graphical representation of the discovered model is shown in *Figure 1* and the mining for the same has been carried out by the Disco process mining tool [8].

*Author for correspondence

Table 1 An event log example

Case ID	Activity	Timestamp
1	A	10:02 AM
1	B	10:04 AM
1	C	10:06 AM
2	A	10:09 AM
2	D	10:13 AM
2	E	10:17 AM
3	A	10:23 AM
3	B	10:28 AM
3	C	12:32 PM
4	A	12:38 PM
4	D	12:45 PM
4	E	12:50 PM
5	A	13:10 PM
5	B	13:26 PM
5	C	13:35 PM

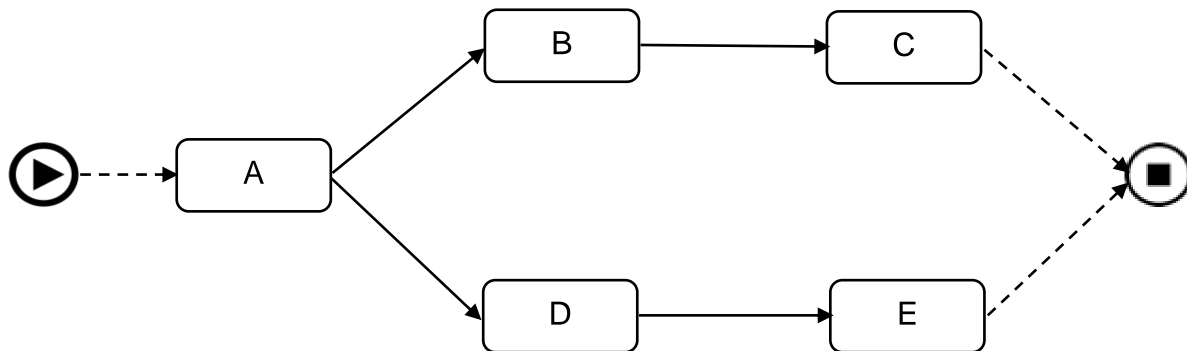

Figure 1 Process model discovered from the events in Table 1

Most of the business organizations nowadays use information technology (IT) as their backbones for execution of their day to day business processes. These IT-enabled organizations produce data in large volumes while executing their business processes. Subsequently, to get a better insight into their executed business processes, they also record the events that are a part of the business process in an event log. Due to the exponential increase in producing data, the event log size for a particular business process also increases rapidly. To do a better analysis of the existing business processes the computationally exhaustive process mining algorithms need to be dealt with effectively. The task becomes challenging as it requires a thorough analysis of the voluminous amount of data available in an event log. Specifically, the process discovery algorithms which are computationally intensive in nature need to be recalibrated so that they can work effectively on ever-increasing event log. One of the effective ways of dealing with this type of situation is parallel processing. Thus, the algorithms available for process discovery at present need to be updated and equipped with the parallel processing features.

1.1Literature review

Process discovery inherits its basis from data mining and business process intelligence. The Alpha algorithm [1] is the oldest algorithm out of several algorithms available for process discovery. The Alpha algorithm considers an event log to be noise free and builds a process model from that event log.

A graphical way to present process models is to display them in the form of block-structured Petri nets. An approach to generate a block-structured Petri nets from a log of recorded events has been proposed in [9]. The approach consisted of two steps. In the first step, an adjacency matrix is built between all the pairs of tasks. The next step involves a deep analysis

of finding the basic structures like sequence, loop, choice, parallel and self-loop to extract the block structured models.

One of the drawbacks of the Alpha algorithm [1] is that it cannot discover the invisible tasks associated with non-free-choice constructs. The authors of [10] developed $\alpha^{\$}$ algorithm to eradicate this problem.

To tackle the problem of process model discovery from very large event log, a divide-and-conquer technique was adopted in the work done in [11]. The approach begins with partitioning a large event log into several smaller event log and then, construct process models for each such small event log. In the next step, all those models are assembled to form the final process model. The objective of the work was to reduce the overall complexity while producing high quality models.

In [12], the authors proposed an integer linear programming (ILP) based method for process discovery. But, the average computation time to solve the ILP problem was too high. Thus, an improvement of this ILP miner was proposed by Van Zelst et al. [13−15]. They emphasized on using regions based on variables. By varying the count of variables (and, hence the regions), the average computation time to solve the ILP problem would be low.

With a limited amount of data available, it is very difficult to build a sound process model and analyze the process behavior. The methods described in [16] and [17] are based on the grammatical inference theory to construct predictive process models from event log. The graphical representations of these predictive models are done through Petri nets. The developed method was named as RegPFA and it was a standalone application.

If the activities in an event log are independent of each other, then they can be executed in parallel. Based on this observation, the authors in [18] proposed an approach to discover process models with concurrency from event log which may not be complete.

Petri nets can be discovered from large event logs by using numerical abstract domains. This concept was used in [19]. The method adopted for implementation of this concept ensures that the discovered models in the form of Petri nets can exactly regenerate all the traces of that log. It also ensures that the log behavior can be represented through these minimal traces.

If an event log does not contain all the execution histories of a process, then the log is treated as an incomplete log. Hence, discovering a correct process model and representing that model with the help of a workflow net becomes challenging. In [20], the authors introduced the concept of invariant occurrence between activities to extract a workflow net from an incomplete log. The approach was based on identifying conjoint occurrence classes of activities. The non-exhibiting behaviors of an event log can be inferred through these conjoint occurrence classes of activities.

Some business processes generate token logs. The data carried through these tokens during a business process execution can not only be helpful in tracking the change of states in such logs, but also, they can be used to enhance the performance of process discovery algorithms. The work done in [21] was based on this theme.

Process model discovery through causal nets were proposed by Greco et al. [22]. The causal relations between activities in an event log is represented in a causal net. The method is based on collecting all the causal relations from an event log and the related knowledge to form the topology for a process model is derived from precedence constraints.

In [23] and [24], Vazquez et al. proposed a genetic algorithm-based method for process model discovery from logs of events. The discovered models were represented through causal nets. The authors developed a standalone miner named as ProDiGen. Like any other genetic algorithm-based approach, this method was also based on the features like fitness, completeness and precision, and operators like crossover and mutation.

The authors in [25] did an extension of the previously available process discovery methods based on causal nets. An optimization method was developed for the causal net output parameters like scalability and interpretability in this work. To analyze a process, first of all, the said process was split into a number of stages and care was taken so that each stage could be mined separately. The objective of the work was to maximize modularity while discovering stage decomposition.

In [26] and [27], the authors proposed a method to discover process models in the form of state machines from event log. They named it CSM Miner. The method focused on different states of a process

and the relationships among those states rather than the events present in a process. Further, a composite state machine was formed using those relations.

Another form of representing the discovered process models is business process model network (BPMN) which was proposed in [28]. In this work, the authors developed a tool named as a BPMN Miner for automated discovery of processes from event logs. The BPMN miner was able to generate BPMN models having loops, activity markers, and sub-processes along with the capability to model exception handling. A subsequent improvement was made to this method in [29] which enables it to deal with the noise present in an event log more effective.

The authors in their work in [30] and [31] proposed approaches to discover declare constraints through the SQLMiner. The standard structured query language (SQL) was used for querying over a relational event log data in [30]. The mining procedure was extremely fast as database performance tuning techniques was applied. Customization of queries and using them from visualizing process perspective beyond control flow was possible [31].

A discovered process model is treated as simple, if it has less branching. Also, the fitness, precision and generalization parameters need to be consistently high and balanced. To discover simple process models an approach had been proposed in [32]. The approach first identifies the splits correctly to record the concurrency, conflict and causal relations among different activities in an event log and then filter out the final directly-follows graph to represent a process model.

Initially, the heuristic miner was proposed in [33] which was able to deal with noises unlike the Alpha algorithm [1]. An improvement to the heuristic miner [33] was proposed in the works [34] and [35]. The tool developed was named as Fodina. Fodina is different from the heuristic miner as the former is more robust in dealing with noises present in an event log. Fodina is also capable of discovering duplicate activities and has flexibility to configure the end user inputs for process discovery.

During process discovery, if recorded infrequent behaviors are considered, then it may lead to the discovery of a complex process model. The method proposed in [36] and [37] is about filtering techniques that deals with removing infrequent behavior from event logs.

Generalization is one of the several parameters available to evaluate the quality of a discovered model. A k-fold cross validation method was proposed in [38] to evaluate the modeled behavior against the observed behavior for an event log on the basis of the generalization parameter.

The method described in [39] is an extension of the inductive miner [40]. Inductive miner extracts process trees from an event log. Discovering process models from event logs containing incomplete traces is a difficult task. The method proposed in [40] can deal with this particular problem.

An enhancement to the heuristic miner [33] was presented in [41]. The method is a two-step approach. In the first step, an accurate but unstructured and unsound process model is discovered. In the following step, a sound structured model is filtered out from that unstructured model.

Till very recent work proposed in [42], the existing literature describes the works relating to process discovery with noisy event log data, dealing with loops in the process model, cluster-based process discovery, decomposing event logs with divide and conquer strategies, enhancing quality of the discovered model, but not focusing on increasing the execution efficiency of the various process discovery algorithms by exploiting parallelism. In [42] the authors have tried to exploit parallelism by detecting independent tasks and running them in parallel in the Alpha algorithm.

1.2Motivation and objective

As efficiency and scalability are the two rudimentary factors to be considered for any type of recalibration of an existing serial algorithm. The motivation behind the proposed work is to investigate these two crucial parameters for the Alpha algorithm in a parallel computing structure that adopts the features available in Open Multi-Processing (OpenMP) API [43]. Specifically, the objective of the current work is to exploit different types of parallelism present in the Alpha algorithm and try to enhance the execution efficiency of the Alpha algorithm by utilizing the features available for parallel computation in the OpenMP library. The present paper proposes to exploit both task-level as well as data-level parallelism to enhance the run-time efficiency of the Alpha algorithm. Hence, a suitable framework has

been developed using the OpenMP API for parallel mode execution of the said algorithm.

Organization of the remaining sections is as follows: section 2 presents the materials and methods like an introduction to various ordering relations available in an event log, a brief introduction on OpenMP and the parallel computing constructs available in OpenMP library, and a meticulous depiction of our proposed framework to utilize the OpenMP library for the parallel execution of the Alpha algorithm. In section 3 the experimental set up, the results and analysis of the results are presented. Section 4 contains the discussion on comparing our results with previously published results in the context of parallelization of the Alpha algorithm. Finally, we present the conclusion and future scope for the improvement in section 5.

2.Materials and methods

2.1Ordering relations in event logs

A run time execution instance of a process essentially contains the following three parameters: the process instance number (i.e. case id), the event name (or the activity that is being performed), and the completion time (or timestamp) of that particular activity. An event log is an enumeration of several such process instances or cases [44]. In a typical business process management system completion of each event, type is synonymous an activity execution. An event log having 5 cases with 15 events is given in *Table 1*. A finite number of events (or activities) within a case collectively constitute a trace.

Definition 1. *(Trace):* A temporally ordered sequence of events or activities present in a process instance, or case can be termed as a trace and let that can be denoted by σ. Hence, $\sigma = t_1 \ldots t_n$.
The traces that constitute the event log given in *Table 1* are:
$\sigma_1 = \sigma_3 = \sigma_5 = <A, B, C>$, and $\sigma_2 = \sigma_4 = <A, D, E>$.

Noise filtering is a critical issue in process mining. Remarkably uncommon traces of process instances in an event log are treated as “outliers” (not “error”) which is further termed as noise [44]. Specifically, in process discovery methods like the Alpha algorithm, non-removal of noise leads to a massively complex process model as the algorithm tries to measure every possible ordering relation. Thus, the Alpha algorithm does not consider noise to build a process model from an event log [1].

Although other algorithms like the flexible heuristic miner algorithm [33, 45, 46] is available to deal with the problem of noise present in an event log and build a process model by considering the same noisy event log, the present work purely focuses on improving the execution efficiency of the Alpha algorithm. The problem of dealing with noise is beyond the scope of this work.

Most of the process discovery algorithms are based on the different categories of ordering or follow relations that exist between the pairs of activities present in an event log. The approach adopted to determine the ordering relations varies from one algorithm to another algorithm. However, before building a process model, all the ordering relations that exist in an event log must be traced out and accumulated. Thus, it is important to record how pair of activities is related to each other and the frequency count of that particular relation of that particular pair of activities rather than count the process instances. A relatively small data set is required to be processed for building a process model once all the ordering relations are computed.

2.2The OpenMP model

The OpenMP application programming interface (API) was developed to facilitate movable shared memory parallel programming [43] in multiprocessor environments. The OpenMP API is well supported with features like appropriate compiler commands, compilation guidance statements and necessary library functions for parallel programming. Even distribution of load and inter-process communication through shared variables are also available to the standard FORTRAN, C, and C++ programming languages through OpenMP API. OpenMP is not only simple, but also fast [47] and is a widely accepted industry standard that exploits data-parallelism as well as task-parallelism [48] through portable multiple-thread shared memory programming.

The shared memory model is a general centralized multi-processing abstract [49] and [47]. It is based on the concept that a number of processors can access and share the common address in memory. OpenMP supports the fork-join multithreading model as shown in *Figure 2* for parallel programming.

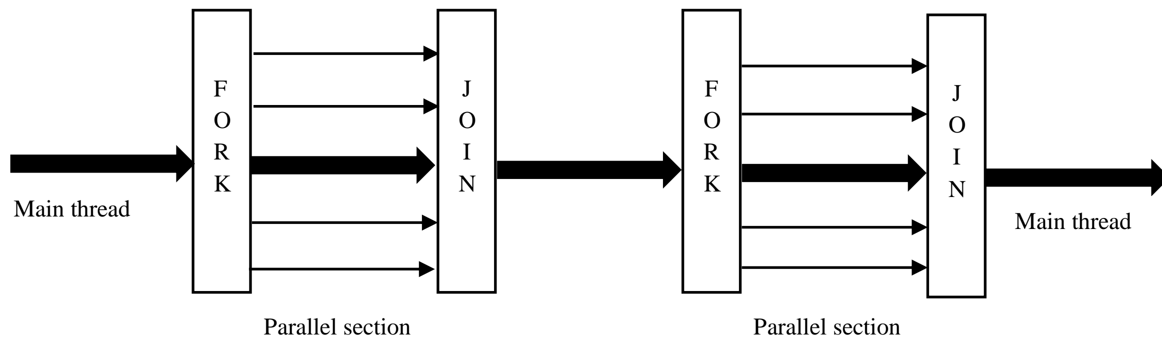

Figure 2 The OpenMP execution model

The program execution begins with a master thread. While the execution of the serial portion is managed by the main thread, the parallel execution is performed by other derived threads. The derived threads are terminated upon the re-execution of the serial part of the program.

2.3The alpha algorithm with the OpenMP framework

The steps of the Alpha algorithm need to be analysed thoroughly before doing any kind of parallel implementation of the Alpha algorithm using the OpenMP API as that would give us a clear view towards parallelizing that algorithm. Thus, in section 2.3.1, first, the Alpha algorithm has been described. As the log-based ordering relationship that exits between a pair activity in an event log is the basis of forming the footprint matrix, special attention was given to it while designing the framework for parallel mode execution of the Alpha algorithm by using the syntaxes available in the OpenMP API in the following section 2.3.2.

2.3.1The alpha algorithm

The Alpha algorithm does not consider noise to be a part of an event log while it tries to generate a process model and subsequently graphically represents that model through a work flow net [1]. All the unique activities present in an event log are found out and grouped into a set in the very first step of the Alpha algorithm [1]. In the subsequent phases, the Alpha algorithm finds out the *causal* relationship [1] derived from different ordering relations existing between various pairs of activities in an event log. Assuming that X and Y are two unique activities from the set of unique activities and it is found that X always precedes Y but not the vice versa. Thus, it can be concluded that there exists a *causal* relation between X and Y. Depending on the ordering relations there can be four categories of *causal* relations that might be existing between a pair of activities in an event log. Definition given in 2 describes these ordering relations.

Definition 2. *(Ordering relations between a pair of activities):* Let P is the event log over a set of unique activities denoted by A. Let the activities $\alpha, \beta \epsilon A$.

- $\alpha >_P \beta$ iff there is a trace $\sigma = t_1t_2t_3...t_n$ and $i \epsilon \{1,...,n\text{-}2\}$ such that $\sigma \epsilon P$, $t_i = \alpha$ and $t_{i+1} = \beta$
- $\alpha \rightarrow_P \beta$ iff $\alpha >_P \beta$ and $\beta \ngtr_P \alpha$
- $\alpha \parallel_P \beta$ iff $\alpha >_P \beta$ and $\beta >_P \alpha$
- $\alpha \#_P \beta$ iff $\alpha \ngtr_P \beta$ and $\beta \ngtr_P \alpha$

The basic temporal ordering relation between a pair of activities is represented by $>_P$. Other ordering relations such as causal, parallel and unrelated are derived from the basic temporal ordering relation. The causal ordering relation between a pair of activities is denoted by $\rightarrow_P$. If two activities are parallel to each other, then the parallel relation can be shown by $\parallel_P$. Two activities are unrelated to each other if neither of them directly follow each other. These unrelated activities are shown through $\#_P$.

The Alpha algorithm generates a footprint matrix using these temporal ordering relationships. Once the footprint matrix has been generated, a process model can be constructed and represented through a work flow net diagram. To have better insights about construction of the footprint matrix, [1] can be referred.

2.3.2The OpenMP implementation of the alpha algorithm

The key to have an effective parallelization of any serial algorithm is to trace out the discrete and independent steps involved in that serial algorithm and execute those steps in parallel. A thorough review of the Alpha algorithm reveals that several steps involved in this algorithm can be run in parallel by adopting either data parallelism or task parallelism [39]. Moreover, the following observations are made

while doing a detailed analysis of the Alpha algorithm:

(i) Step 1 of the Alpha algorithm is to find all the unique or distinct activities available in an event log and group them into a set denoted by A_P. This step can be solved in data parallel approach. The parallel for directive available in the OpenMP API can be used to achieve loop-level parallelism to exploit data-parallelism available in the first step of the Alpha algorithm.

(ii) Step 2 and Step 3 of the Alpha algorithm are associated with finding out the set of all start activities and end activities denoted by A_S and A_E respectively from the event log. The same divide-and-conquer approach considered in step 1 can be adopted for the calculation of these two steps too. For these two steps, again we can take help of the parallel for directive available in the OpenMP API to exploit data parallelism.

(iii) As computation of A_P, A_S and A_E are independent of each other, these steps can be run in parallel by exploiting task parallelism. To achieve this, we can take help of the section directive available in the OpenMP API.

(iv) To generate the footprint matrix, we need to check for the four types of ordering relations defined in section 2.3.1 for every pair of activities. Thus, each activity needs to be compared with every other activity to deduce a specific relation between them. The upper bound for these comparisons is limited by the count of unique or distinct activities discovered from the event log. We can consider setting the number of threads equals to the number of unique activities and using the parallel for directive available in the OpenMP API to generate the footprint matrix.

(v) Step 5 and Step 6 of the Alpha algorithm deals with computing the set pairs and maximal set pairs denoted by R_P and Q_P respectively. Once the footprint matrix is generated, a task-parallel approach can be considered for executing Step 5 and Step 6 with the help of sections directive available in the OpenMP API.

It is significant to note that the count of distinct activities fetched from an event log derives the rest of the steps of the Alpha algorithm rather than the number of event entries made into the event log. Hence, employing a parallel execution framework using the OpenMP API to execute the preliminary steps of the Alpha algorithm which involves finding out the unique or distinct activities and tracing out the different categories of ordering relations that exist among those activities, becomes important to get benefited from the parallel computation.

Considering the above observations, a framework for parallel execution of the Alpha algorithm can be developed as the schematic representation given in *Figure 3*.

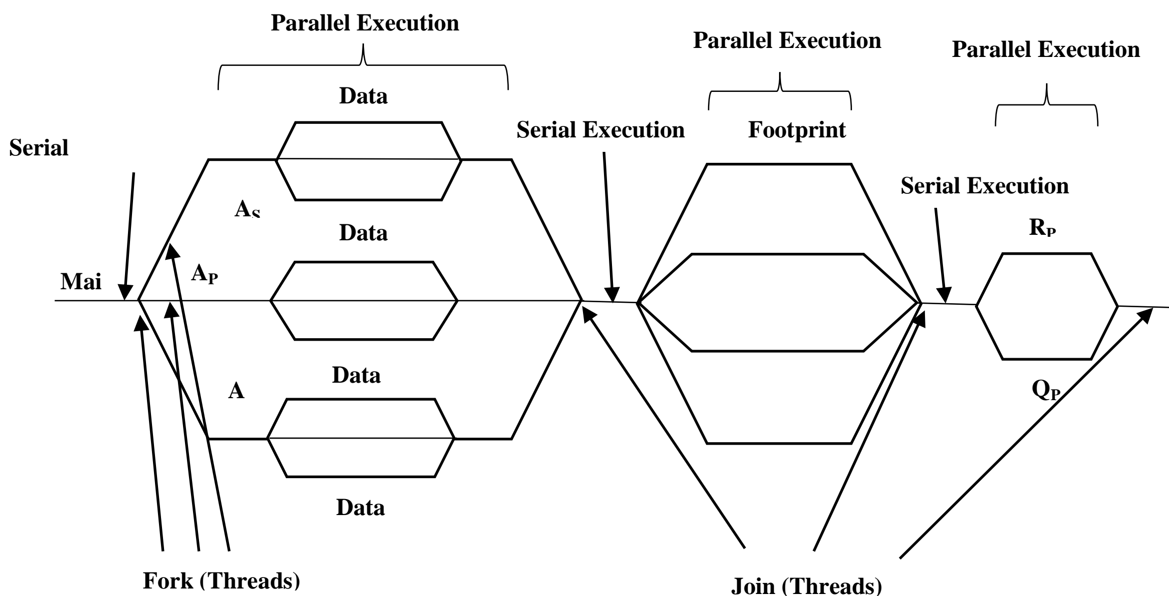

Figure 3 A framework for parallel execution of the Alpha algorithm

Pertaining to the parallel execution framework given in *Figure 3*, the pseudocode to implement the framework using the OpenMP API can be given as:

```
int main()
    {//preprocessing steps
    # pragma omp parallel{
    #pragma omp sections{
    #pragma omp section{
    #pragma omp parallel for
    Compute A_P;}
    #pragma omp section{
    #pragma omp parallel for
    Compute A_S;}
    #pragma omp section{
    #pragma omp parallel for
    Compute A_E;}} }
    //intermediate steps
    #pragma omp parallel for
    Generate footprint matrix;
    //intermediate steps
    #pragma omp parallel{
    #pragma omp sections{
    #pragma omp section{
    Compute R_P;}
    #pragma omp section{
    Compute Q_P;} } }
    /* Execute rest of the steps and draw the
    workflow net */
    return 0; }
```

3.Results

For the purpose of experiments, we implemented the Alpha algorithm in the serial form using the C++ language. The parallel version was also implemented using the C++ language with the help of OpenMP 3.0 API. The GCC 4.4.6. was used to develop both the versions. All the program executions related to the experiments were carried out in one of the computing nodes available in a high-performance computing cluster (HPCC). The HPCC contains 13 nodes in total. Each node in that HPCC is an Intel Xeon E5-2650 machine having dual CPU's and 64 GB of RAM. Within a node, each CPU was having 8 computing cores working at a base frequency of 2.0 GHz. Thus, physically 16 computing cores were available for carrying out the program executions. The HPCC is equipped with Red Hat Linux 6.3 64-bit operating system. During the program execution, the hyper threading was set on.

The runtime impact of the parallel mode execution of the Alpha algorithm was measured through several experiments. To visualize the real impact of a parallel computing framework: either the computation need to be massively complex in nature or the computation is based on large volume of data. Thus, our initial objective was to find out the real-world event logs of very large size. But these were not available. Hence, we generated synthetic event logs of large size through a home-grown small tool developed using C programming language. Specifically, we created comma-separated-values (CSV) files to represent event log files. Each entry in those synthetic event logs was having the following three fields: a process instance number (or case id), a reference to the activity that has been performed (or the activity name), and the completion time of that activity (or the time stamp of each activity).

The activity names were strings of variable lengths rather than characters. Parameters that would have affected the execution of the Alpha algorithm such as the number of unique activities and the number of different types of splits were considered while generating the synthetic event logs using our home-grown tool. We created five process models by setting the number unique activities equal to 5, 10, 15, 20 and 25 respectively. Each process model contained at least one AND split and one XOR split. Similarly, we considered the number of unique cases as 3, 8, 13, 18 and 23 respectively, for the process models having number of unique activities 5, 10, 15, 20, and 25 respectively and care was taken to consider all the variants of cases possible for a particular process model. We made near about 4×10^7 event entries for each individual event log. Thus, the generated event log files were approximately 2 GB large to have an effective parallel computation. For all the programs, the execution times were recorded by considering average of 100 runs. Readings in *Table 2* shows the average execution time of the Alpha algorithm when run in serial mode.

To have a near-to-equal allocation of threads for independent execution of tasks, we manually set 12, 10, 10 threads to compute A_P, A_S, and A_E respectively in the parallel mode execution of the Alpha algorithm. The readings in *Table 2* indicate that the task of finding out A_P from an event log takes more time when compared to the tasks of finding out A_S and A_E. Thus, we decided to allocate a slightly higher count of threads for the execution of the step A_P than the steps A_S and A_E. The tasks like computing the four temporal ordering relations defined in section 2.3.1 deals with comparing each activity with every other activity available in the event log and the upper bound of these comparisons were limited by the

number of unique activities available in the event log. Thus, we manually set the number of threads equal to the number of unique activities for computation of different temporal ordering relations which subsequently generates the footprint matrix. Similarly, for the computation of R_P and Q_P, parallel sections with the number of threads set to two were considered. With this thread set-up, on applying the parallel for loop as well as the parallel sections syntax available in the OpenMP API, we achieved a noticeable reduction in execution time of the Alpha algorithm. Here also, we recorded an average of 100 runs for each case of parallel execution. The readings related to the average execution time in parallel mode of the Alpha algorithm are presented in *Table 3*.

Table 2 Serial mode execution time of the Alpha algorithm

No. of unique activities	Individual module execution time in seconds						Total execution time in seconds
	A_P	A_S	A_E	Footprint matrix	R_P	Q_P	
5	15.94	0.00014	0.00015	148.96	4.32	5.65	174.8703
10	25.89	0.00018	0.00018	448.87	6.18	7.23	488.1704
15	35.16	0.00023	0.00022	920.98	9.12	10.78	976.0405
20	42.91	0.00029	0.00028	1562.95	23.26	15.11	1634.231
25	50.72	0.00035	0.00034	2352.92	20.54	22.83	2447.011

Table 3 Parallel mode execution time of the Alpha algorithm

No. of unique activities	Individual module execution time in seconds			Total execution time in seconds
	A_P, A_S, A_E	Footprint matrix	R_P, Q_P	
5	1.58	29.792	5.71	37.082
10	2.62	44.887	7.36	54.867
15	3.49	61.398	10.92	75.808
20	4.01	104.196	15.19	123.396
25	4.88	158.981	23.11	186.971

The *speedup* [50] parameter plays an important role in determining the performance improvement due to any kind of enhancement incorporated into an existing computational method. Thus, in our case also, we considered *speedup* factor to show the extent of performance improvement we achieved through the parallel execution of the Alpha algorithm. The *speedup* Equation can be given as follows:

$$Speedup(S) = \left(\frac{Told}{Tnew}\right) \quad (1)$$

where T_{old} represents the recorded execution time without any improvement or serial mode execution time and T_{new} is the recorded new execution time with improvement or parallel mode execution time [42].

For this experimental set up:
T_{old} = Average Execution Time in Serial mode = T_s and T_{new} = Average Execution Time in Parallel mode = T_p.
Hence,

$$Speedup(S) = \left(\frac{Tserial\ execution\ time}{Tparallel\ execution\ time}\right) = \left(\frac{Ts}{Tp}\right) \quad (2)$$

The base *speedup* was set to *1x* by considering the serial mode execution time of the Alpha algorithm. *Table 4* shows the *speedup* values that we were able to get with different number of distinct activities.

Table 4 Speedup achieved

No. of unique activities	T_s in seconds	T_p in seconds	Speedup
5	174.8703	37.082	4.71
10	488.1704	54.867	8.89
15	976.0405	75.808	12.87
20	1634.231	123.396	13.24
25	2447.011	186.971	13.08

4.Discussion

It can be observed from *Table 4* that the speedup is increasing as we kept on increasing the number of threads. This is obvious as the generation of footprint matrix, which is the most time-consuming step in the entire Alpha algorithm, relies on the number of unique activities and the temporal ordering relations existing among those unique activities. A significant reduction in execution time of the Alpha algorithm was recorded as the computation of the footprint

matrix was done in parallel mode by launching multiple threads equal to the number of unique activities extracted from an event log.

The theoretical upper limit of the *speedup* factor in the work done in [42] is *4x* as only independent and discrete tasks have been identified and run on different computing nodes. Given the program structure in [42], maximum four independent tasks can run in parallel which leads to a maximum theoretical *speedup* of *4x*. But the present work successfully exploits data as well as task parallelism in the Alpha algorithm. Most importantly, the performance indicator in terms of the *speedup* is not limited by the number of independent tasks for the present work. Rather, the *speedup* is limited by the number of unique activities. Hence, a maximum theoretical *speedup* of *25x* could be possible subject to availability of such number of physical threads. However, with the present set up we were able to get a maximum *speedup* of *13.24x*, which outperforms the *speedup* achieved in [42]. Another noticeable thing is that the task-level parallelism approach taken in [42] was implemented through a message passing interface (MPI) programming concept which suffers from overheads generated due to communication latency. But the present approach has zero or negligible communication latency as it is based on shared memory multiprocessing model.

The proposed parallel framework based on the OpenMP library can now be adopted for computation of the different modules available in the Alpha algorithm. Tracing out the number of unique activities and building a footprint matrix by computing different types of ordering relations observed in an event log can further be applied effectively for large event logs and higher count of unique activities.

5.Conclusion

The performances of the Alpha algorithm in parallel mode of execution were observed through a series of experiments. The construct and syntaxes available in the OpenMP library for parallel computing were used for execution of different modules of the Alpha algorithm. It can be observed that the highest *speedup* achieved is *13.24x*. The noticeable thing is that the speedup factor relies on the number of threads. The overall execution time of the Alpha algorithm in parallel mode was significantly less in comparison to the serial mode as the number threads launched varied according to the number of unique activities. Observations from the readings taken during the experiments also indicate that the parallel mode of execution outruns the serial mode of execution of a larger log data. Thus, an accelerated computation of the Alpha algorithm was possible by using the proposed parallel framework based on centralized shared memory computing available in the OpenMP API.

The significant advantages of using the OpenMP API are: (i) the memory access time remains uniform as a centralized memory is shared among multiple processors or cores, (ii) portable multithreading code, and (iii) a relatively high-level abstraction is available to write parallel programs. The present work is limited to a computing environment characterized by multi-core processors and shared memory architecture. The biggest problem with this type of architecture is that the memory system may not be able to support the bandwidth demand beyond a certain number of processors or cores. We propose to extend our work to a computing environment characterized by multi-node-multi-core architecture with hybrid OpenMP-MPI programming framework in a high-performance computing cluster.

Acknowledgment
None.

Conflicts of interest
The authors have no conflicts of interest to declare.

References

[1] Van Der Aalst W, Weijters T, Maruster L. Workflow mining: discovering process models from event logs. IEEE Transactions on Knowledge & Data Engineering. 2004; 16(9):1128-42.
[2] Kindler E, Rubin V, Schäfer W. Process mining and petri net synthesis. In international conference on business process management 2006 (pp. 105-16). Springer, Berlin, Heidelberg.
[3] Bergenthum R, Desel J, Lorenz R, Mauser S. Process mining based on regions of languages. In international conference on business process management 2007 (pp. 375-83). Springer, Berlin, Heidelberg.
[4] Motahari-Nezhad HR, Saint-Paul R, Benatallah B, Casati F. Deriving protocol models from imperfect service conversation logs. IEEE Transactions on Knowledge and Data Engineering. 2008; 20(12):1683-98.
[5] Bergenthum R, Desel J, Mauser S, Lorenz R. Construction of process models from example runs. In transactions on petri nets and other models of concurrency II 2009 (pp. 243-59). Springer, Berlin, Heidelberg.
[6] Solé M, Carmona J. Region-based foldings in process discovery. IEEE Transactions on Knowledge and Data Engineering. 2013; 25(1):192-205.

[7] Van Dongen BF, De Medeiros AA, Wen L. Process mining: overview and outlook of petri net discovery algorithms. In transactions on petri nets and other models of concurrency II 2009 (pp. 225-42). Springer, Berlin, Heidelberg.
[8] Günther CW, Rozinat A. Disco: discover your processes. BPM (Demos). 2012; 940:40-4.
[9] Huang Z, Kumar A. A study of quality and accuracy trade-offs in process mining. INFORMS Journal on Computing. 2012; 24(2):311-27.
[10] Guo Q, Wen L, Wang J, Yan Z, Philip SY. Mining invisible tasks in non-free-choice constructs. In international conference on business process management 2015 (pp. 109-25). Springer, Cham.
[11] Verbeek HM, Van Der Aalst WM, Munoz-Gama J. Divide and conquer: a tool framework for supporting decomposed discovery in process mining. The Computer Journal. 2017; 60(11):1649-74.
[12] Van DerWerf JM, Van Dongen BF, Hurkens CA, Serebrenik A. Process discovery using integer linear programming. Fundamenta Informaticae. 2009; 94(3-4):387-412.
[13] Van Zelst SJ, Van Dongen BF, Van Der Aalst WM, Verbeek HM. Discovering workflow nets using integer linear programming. Computing. 2018; 100(5):529-56.
[14] Van Zelst SJ, Van Dongen BF, Van Der Aalst WM. Avoiding over-fitting in ILP-based process discovery. In international conference on business process management 2016 (pp. 163-71). Springer, Cham.
[15] Van Zelst SJ, Van Dongen BF, Van Der Aalst WM. ILP-based process discovery using hybrid regions. In ATAED@ petri nets/ACSD 2015 (pp. 47-61).
[16] Breuker D, Matzner M, Delfmann P, Becker J. Comprehensible predictive models for business processes. MIS Quarterly. 2016.
[17] Breuker D, Delfmann P, Matzner M, Becker J. Designing and evaluating an interpretable predictive modeling technique for business processes. In international conference on business process management 2014 (pp. 541-53). Springer, Cham.
[18] Song W, Jacobsen HA, Ye C, Ma X. Process discovery from dependence-complete event logs. IEEE Transactions on Services Computing. 2016; 9(5):714-27.
[19] Carmona J, Cortadella J. Process discovery algorithms using numerical abstract domains. IEEE Transactions on Knowledge and Data Engineering. 2014; 26(12):3064-76.
[20] Tapia-Flores T, Rodríguez-Pérez E, López-Mellado E. Discovering process models from incomplete event logs using conjoint occurrence classes. In ATAED@ petri nets/ACSD 2016 (pp. 31-46).
[21] Li C, Ge J, Huang L, Hu H, Wu B, Yang H, et al. Process mining with token carried data. Information Sciences. 2016; 328:558-76.
[22] Greco G, Guzzo A, Lupia F, Pontieri L. Process discovery under precedence constraints. ACM Transactions on Knowledge Discovery from Data. 2015; 9(4).
[23] Vázquez-Barreiros B, Mucientes M, Lama M. ProDiGen: mining complete, precise and minimal structure process models with a genetic algorithm. Information Sciences. 2015; 294:315-33.
[24] Vázquez-Barreiros B, Mucientes M, Lama M. A genetic algorithm for process discovery guided by completeness, precision and simplicity. In international conference on business process management 2014 (pp. 118-33). Springer, Cham.
[25] Nguyen H, Dumas M, Ter Hofstede AH, La Rosa M, Maggi FM. Mining business process stages from event logs. In international conference on advanced information systems engineering 2017 (pp. 577-94). Springer, Cham.
[26] Van Eck ML, Sidorova N, Van Der Aalst WM. Discovering and exploring state-based models for multi-perspective processes. In international conference on business process management 2016 (pp. 142-57). Springer, Cham.
[27] Van Eck ML, Sidorova N, Van Der Aalst WM. Guided interaction exploration in artifact-centric process models. IEEE conference on business informatics 2017 (109-18). IEEE.
[28] Conforti R, Dumas M, García-Bañuelos L, La Rosa M. Beyond tasks and gateways: discovering BPMN models with subprocesses, boundary events and activity markers. In international conference on business process management 2014 (pp. 101-17). Springer, Cham.
[29] Conforti R, Dumas M, García-Bañuelos L, La Rosa M. BPMN miner: automated discovery of BPMN process models with hierarchical structure. Information Systems. 2016; 56:284-303.
[30] Schönig S, Rogge-Solti A, Cabanillas C, Jablonski S, Mendling J. Efficient and customisable declarative process mining with SQL. In international conference on advanced information systems engineering 2016 (pp. 290-305). Springer, Cham.
[31] Schönig S, Di Ciccio C, Maggi FM, Mendling J. Discovery of multi-perspective declarative process models. In international conference on service-oriented computing 2016 (pp. 87-103). Springer, Cham.
[32] Augusto A, Conforti R, Dumas M, La Rosa M. Split miner: discovering accurate and simple business process models from event logs. International conference on data mining 2017 (pp. 1-10). IEEE.
[33] Weijters AJ, Van Der Aalst WM. Rediscovering workflow models from event-based data using little thumb. Integrated Computer-Aided Engineering. 2003; 10(2):151-62.
[34] Vanden Broucke SK, De Weerdt J. Fodina: a robust and flexible heuristic process discovery technique. Decision Support Systems. 2017; 100:109-18.
[35] De Weerdt J, Vanden Broucke SK, Caron F. Bidimensional process discovery for mining BPMN models. In international conference on business process management 2014 (pp. 529-40). Springer, Cham.

[36] Conforti R, La Rosa M, Ter Hofstede AH. Filtering out infrequent behavior from business process event logs. IEEE Transactions on Knowledge and Data Engineering. 2017; 29(2):300-14.
[37] Mannhardt F, De Leoni M, Reijers HA, Van Der Aalst WM. Data-driven process discovery-revealing conditional infrequent behavior from event logs. In international conference on advanced information systems engineering 2017 (pp. 545-60). Springer, Cham.
[38] Van Dongen B, Carmona J, Chatain T, Taymouri F. Aligning modeled and observed behavior: a compromise between computation complexity and quality. In international conference on advanced information systems engineering 2017 (pp. 94-109). Springer, Cham.
[39] Leemans M, Van Der Aalst WM. Modeling and discovering cancelation behavior. In OTM confederated international conferences" on the move to meaningful internet systems" 2017 (pp. 93-113). Springer, Cham.
[40] Leemans SJ, Fahland D, Van Der Aalst WM. Discovering block-structured process models from event logs containing infrequent behaviour. In international conference on business process management 2013 (pp. 66-78). Springer, Cham.
[41] Augusto A, Conforti R, Dumas M, La Rosa M, Bruno G. Automated discovery of structured process models from event logs: the discover-and-structure approach. Data & Knowledge Engineering. 2018; 117:373-92.
[42] Sahu M, Chakraborty R, Nayak G. A task-level parallelism approach for process discovery. International Journal of Engineering & Technology. 2018: 7(4):2446-52.
[43] Ciorba FM, Iwainsky C, Buder P. OpenMP loop scheduling revisited: making a case for more schedules. In international workshop on OpenMP 2018 (pp. 21-36). Springer, Cham.
[44] Van Der Aalst WM. Process mining: data science in action. Springer; 2016.
[45] Weijters AJ, Van Der Aalst WM, De Medeiros AA. Process mining with the heuristics miner-algorithm. Technische Universiteit Eindhoven, Tech. Rep. WP. 2006; 166:1-34.
[46] Weijters AJ, Ribeiro JT. Flexible heuristics miner (FHM). In symposium on computational intelligence and data mining 2011 (pp. 310-7). IEEE.
[47] Serrano MA, Royuela S, Quiñones E. Towards an OpenMP specification for critical real-time systems. In international workshop on OpenMP 2018 (pp. 143-59). Springer, Cham.
[48] Pacheco P. An introduction to parallel programming. Elsevier; 2011.
[49] Kemp J, Chapman B. Mapping OpenMP to a distributed tasking runtime. In international workshop on OpenMP 2018 (pp. 222-35). Springer, Cham.
[50] Hennessy JL, Patterson DA. Computer architecture: a quantitative approach. Elsevier; 2011.

Muktikanta Sahu is working as an Assistant Professor in the Department of Computer Science & Engineering at IIIT Bhubaneswar. He has completed his B.Tech. and M.Tech. in Computer Science & Engineering from Biju Pattanaik University of Technology, Odisha in 2003 and 2007 respectively. His current research projects are on Parallel Computing and Process Mining. He has published his past research work in IEEE Transaction on Neural Networks and International Journal of Engineering & Technology.
Email: muktikanta@iiit-bh.ac.in

Dr. Gopal Krishna Nayak has graduated from Indian Institute of Technology Kharagpur and Indian Institute of Management Bangalore. Presently he is a professor in the Department of Computer Science & Engineering at IIIT Bhubaneswar. He has nearly 30 years of teaching and research experience. He is also an advisor to the Government of Odisha on adoption of technology. He has nearly 30 years of teaching and research experience.
Email: gopal@iiit-bh.ac.in