Vidushi: Parallel Implementation of Alpha Miner Algorithm and Performance Analysis on CPU and GPU Architecture

Divya Kundra[1], Prerna Juneja[1], and Ashish Sureka[2(✉)]

[1] Indraprastha Institute of Information Technology, Delhi (IIITD), New Delhi, India
[2] Software Analytics Research Lab (SARL), New Delhi, India
ashish@iiitd.ac.in
http://www.iiitd.ac.in/
http://www.software-analytics.in/

Abstract. Process Aware Information Systems (PAIS) are IT systems which support business processes and generate event-logs as a result of execution of the supported business processes. Alpha Miner is a popular algorithm within Process Mining which consists of discovering a process model from the event-logs. Discovering process models from large volumes of event-logs is a computationally intensive and a time consuming task. In this paper, we investigate the application of parallelization on Alpha Miner algorithm. We apply implicit multithreading parallelism and explicit parallelism through `parfor` on it offered by MATLAB (Matrix Laboratory) for multi-core Central Processing Unit (CPU). We measure performance gain with respect to serial implementation. Further, we use Graphics Processor Unit (GPU) to run computationally intensive parts of Alpha Miner algorithm in parallel. We achieve highest speedup on GPU reaching till 39.3× from the same program run over multi-core CPU. We conduct experiments on real world and synthetic datasets.

Keywords: Alpha miner algorithm · GPU · MATLAB · Multi-core CPU · Parallel Computing Toolbox (PCT) · Parallel programming · PAIS

1 Research Motivation and Aim

Process Mining consists of analyzing event-logs generated by PAIS for the purpose of discovering run-time process models, checking conformance between design-time and run-time process maps, analyzing the process from control flow and organizational perspective for the purpose of process improvement and enhancement [1]. Performance improvement of computationally intensive Process Mining algorithms is an important issue due to the need to efficiently process the exponentially increasing amount of event-logs data generated by PAIS. Distributed and Grid computing, parallel execution on multi-core processors and

M. Reichert and H.A. Reijers (Eds.): BPM Workshops 2015, LNBIP 256, pp. 230–241, 2016.
DOI: 10.1007/978-3-319-42887-1_19

using hardware accelerators such as GPU are well-known solution approaches for speeding-up the performance of data mining algorithms.

Alpha Miner algorithm is one of the fundamental algorithms in Process Mining for discovering a process model (reconstructing causality and workflow between activities) from event-logs consisting of process instances or traces [2]. Our analysis of the Alpha Miner algorithm reveals that the algorithm contains independent tasks which can be split among different workers or threads and thus we believe that the algorithm has the ability or property of parallelization. The work presented in this paper is motivated by the need to reduce the execution time of Alpha Miner algorithm on multi-core CPU and GPU based hardware accelerators. The research aim of the work presented in this paper is as follows:

1. To propose a parallel approach for Alpha Miner algorithm by designing a decomposition strategy for partitioning the workload across multiple cores on CPU through implicit and explicit parallelism provided by MATLAB.
2. To perform parallelization of Alpha Miner algorithm on a GPU and examine the extent of speedup due to the hardware accelerator.
3. To investigate the efficiency and performance gain of different types of parallelisms (implicit, explicit, GPU) on Alpha Miner algorithm by conducting a series of experiments on both real world and synthetic datasets.

2 Related Work and Research Contributions

Implementation of data mining algorithms on multi-core CPU and GPU processors is an area that has attracted several researchers attention. Ahmadzadeh et al. [3] present a parallel method for implementing k-NN (k-nearest neighbor) algorithm in multi-core platform and tested their approach on five multi-core platforms demonstrating best speedup of 616×. Arour et al. [4] present two FP-growth (Frequent Pattern) implementations that takes advantage of multi-core processors and utilize new generation GPUs. Lu et al. [12] develop a method which adopts the GPU as a hardware accelerator to speed up the sequence alignment process. Ligowski et al. [11] uses CUDA programming environment on Nvidia GPUs and ATI Stream Computing environment on ATI GPUs to speed up the Smith Waterman sequence alignment algorithm. Their implementation strategy achieves a 3.5× higher per core performance than the previous implementations of this algorithm on GPU [11]. In context to existing work and to the best of our knowledge, the study presented in this paper makes the following novel contributions:

1. A parallel implementation of Alpha Miner algorithm and an in-depth study (with several real and synthetic dataset) on improving the execution performance by using multi-core CPU.
2. A focused study on accelerating Alpha Miner algorithm through parallelism on GPU and testing the approach on various real and synthetic datasets.

3 Research Framework and Solution Approach

In sequential programming, there is an ordered relationship of execution of instructions where only a single instruction executes at a particular instance of time. On the contrary, parallel programming lets execution of multiple tasks at the same instance of time by distributing work to different processors which run in parallel [10]. The Alpha Miner algorithm [2] starts with finding the direct succession relation by scanning the event-logs. For activities 'x' and 'y' if activity 'y' occurs immediately after activity 'x' in log trace direct succession relation holds on 'xy'. Causal (direct succession holds on 'xy' but not on 'yx'), parallel (direct succession holds both on 'xy' and 'yx') and unrelated relations (direct succession holds neither on 'xy' nor on 'yx') are determined through direct succession relation between every two activities and stored in a footprint matrix. All pair of sets (A,B) (A and B are sets containing distinct activities) are found such that all activities within set A and B are unrelated to each other whereas every activity in set A has causal relation with every activity of set B. All activities in A are connected to arcs directed to a place (represented by circle which symbolises conditions) and from the place arcs are directed to all activities in set B. Pair of sets (A,B) that connect maximum activities through a single place are only retained as maximal set pairs. Set of initial and final activities which are detected for connecting to a initial and final place respectively along with pair of maximal sets connected through a place represent the Petri Net [7] output of Alpha Miner. The algorithm can be broken into discrete and independent tasks which can be solved concurrently. We implement parallelization on single-threaded version of Alpha Miner algorithm by 3 kinds of parallelism supported by MATLAB. For all the implementations, we encode activity names in the input event-log by unique positive integers.

3.1 Sequential Single Threading on CPU

A single-threaded program runs sequentially. Serial implementation done on a single thread provides a base for evaluating comparisons from other implementations (multi-threaded, `parfor`). To prevent the trigger of implicit multi-threading by MATLAB, we enable only a single thread in the program by using `maxNumCompThreads(1)`. We use none of the inbuilt multi-threaded functions[1] available in MATLAB for single-threaded implementation. As shown in Fig. 1(a) the main functionalities in Alpha Miner algorithm like determining all the direct succession relations by scanning the entire event- log, building the footprint matrix and determining the maximal set pairs are implemented through `for` loop. Use of `for` loops makes the program work sequentially and slower. At each iteration of `for` loop conditions are checked and branching occurs adding to more overheads and affecting the code performance.

[1] http://www.mathworks.com/matlabcentral/answers/95958-which-matlab-functions-benefit-from-multithreaded-computation.

```
for i=1:traces
%Find DirectSuccession
end

for i=1:activities
%Build Footprint
end

for i=1:activities
%Find Maximal Set Pairs
end
```

(a) Single-threaded implementation.

```
parfor i=1:traces
%Find DirectSuccession
end

parfor i=1:activities
%Build Footprint
end

parfor i=1:activities
%Find Maximal Set Pairs
end
```

(b) `parfor` implementation.

```
[m n]=size(InputFile);
ShiftedFile=InputFile(1:m,2:n);
DirectSuccession=arrayfun(@CantorPairing,InputFile,ShiftedFile);
```

(c) Multi-threaded implementation.

Fig. 1. Fragments of alpha miner algorithm implementation in MATLAB Code showing programming constructs for Multi-core CPU and GPU implementations.

3.2 Explicit Parallelism on CPU

MATLAB has Parallel Computing Toolbox (PCT)[2] for applying external parallelism over set of independent tasks. `parfor`[3] in PCT allows execution of the loop iterations in parallel on workers. Workers are threads that are executed on processor cores. Using `parfor`, a separate process is created for each worker having its own memory and CPU usage. There are communication overheads associated with setting the workers and copying the data to each of them.

When `parfor` is executed, the MATLAB client coordinates with the MATLAB workers which form a parallel pool. The code within the `parfor` loop is distributed to workers executing in parallel in the pool and the results from all the workers are collected back by the client[4]. The body of the `parfor` is an iteration which is executed in no particular order by the workers. Thus the loop iterations are needed to be independent of each other. If number of iterations equals the number of workers in the parallel loop, each iteration is executed by one worker else a single worker may receive multiple iterations at once to reduce the communication overhead[5]. To start a pool of workers `parpool` (profilename, poolsize)[6] is used where name of the parallel pool forming a cluster is to be specified in the 'profilename' and size of the cluster in the 'poolsize' argument.

[2] http://in.mathworks.com/products/parallel-computing/.
[3] http://in.mathworks.com/help/distcomp/parfor.html.
[4] http://cn.mathworks.com/help/pdf_doc/distcomp/distcomp.pdf.
[5] http://in.mathworks.com/help/distcomp/introduction-to-parfor.html.
[6] http://in.mathworks.com/help/distcomp/parpool.html?refresh=true.

We identify following 3 `for` loops (amongst several `for` loops within the algorithm) executing the main functionalities of the algorithm and also containing the code body that is independent at each iteration (a condition for parallel execution) enabling us to apply `parfor`:

1. **Determining direct succession relation**: The task of discovering the pair of activities having direct succession relation can be distributed to different workers with first worker calculating the direct succession relations from one trace, second worker from some other trace and so on. As shown in Fig. 1(b), first `parfor` loop distributes the total 'traces' present in input file among various workers. Results can be gathered from each worker, redundancies can be removed and unique pair of activities having direct succession between them can be deduced.
2. **Building up the footprint matrix**: The process of creating the footprint can be broken into independent work of finding all different relations (causal, paralllel, unrelated) of an activity with the rest of the activities by a worker. In second `parfor` loop of Fig. 1(b), each worker upon receiving a activity from total unique 'activities' computes all relations of the received activity with the remaining activities.
3. **Forming maximal set pairs**: The function to discover maximal set pairs can also be broken into smaller independent tasks of determining all the maximal set pairs (A,B) by a worker that can be formed by including a particular activity in set A. In third `parfor` loop of Fig. 1(b) a worker on receiving an activity from total unique 'activities', computes all the possible maximal set pairs (A,B) that can be formed by including the received activity in set A. With each worker doing the same simultaneously, we can gather the results faster and after removing redundancies we can get distinct maximal set pairs.

Through experiments we observe that both the footprint matrix building and maximal set pair generation do not consume much time (less than 1% of program's execution time) in single-threaded implementation. Whereas calculating direct succession incurs about 90% of program's execution time. The majority of the program's execution time incurred towards computing direct succession is because in most real world datasets the count of activities is less than the number of traces to be scanned for determining direct succession by several orders of magnitude. Thus, calculating direct succession is the bottleneck for the program and bringing the benefits of parallelization to it can help in attaining a good speedup.

3.3 Multithreading Parallelism on CPU

In MATLAB by default implicit multithreading[7] is provided for functions and expressions that are combinations of element wise operations. In this type of parallelism, multiple instruction streams are generated by one instance of MATLAB

[7] http://www.mathworks.com/matlabcentral/answers/95958-which-matlab-functions-benefit-from-multi-threaded-computation.

session that are accessed by multiple cores[8]. To achieve implicit multithreading each element wise operation should be independent of each other. Size of data should be big enough so that speedup achieved by the concurrent execution exceeds the time required for partitioning and managing different threads. To fulfil these requirements and thus implicit parallelism, vectorization[9] of independent and big tasks is essential. Vectorization is one of the most efficient ways of writing the code in MATLAB [8]. It performs operations on large matrices through a single command at once instead of performing each operations one by one inside the `for` loop. An effective way of applying vectorization is replacing `for` loops by vector operations. Code using vectorization uses optimised multi-threaded linear algebra libraries and thus generally run faster than its counterpart `for` loop [8]. The determination of the bottleneck direct succession relation can be vectorized using `arrayfun`[10]. MATLAB uses implicit multithreading using commands such as- `arrayfun`. We use `arrayfun` for performing element wise operations on input matrices. `arrayfun`(func,A1,...,An) applies the function specified in function handle 'func' to each element of equal sized input arrays. The order of execution of function on the elements is not specific, thus tasks should be independent of each other. Figure 1(c) shows the implementation of `arrayfun` in the algorithm in which the 'ShiftedFile' argument to `arrayfun` is the input event-log file ('InputFile') shifted to left by 1. Each cell of the 'ShiftedFile' contains the immediate succeeding activity of the activity present in the corresponding cell of 'InputFile'. Thus direct succession relation holds between corresponding cells of the 'InputFile' and 'ShiftedFile'. We apply Cantor pairing function[11] [5,6] in 'func' as shown in Fig. 1(c) on each two corresponding elements of the input matrices by which pair of activities having direct succession relation are uniquely encoded and stored.

3.4 Parallelism on GPU

While a CPU has a handful number of cores, GPU has a large number of cores along with dedicated high speed memory [9]. GPUs perform poor when given a piece of code that involves logical branching. They are meant for doing simple scalar arithmetic (addition, subtraction, multiplication, division) tasks by hundreds of threads running in parallel [13]. GPU can be accessed by MATLAB through Parallel Computing Toolbox[12]. We can offload discovering of direct succession relation which consumes major part in the running time of the algorithm as discussed in Sect. 3.2 to GPU. Computation of direct succession does not involve much of branching across its code, can be transformed into element wise operations and its computation time far exceeds the transfer time to and from

[8] http://in.mathworks.com/company/newsletters/articles/parallel-matlab-multiple-processors-and-multiple-cores.html.
[9] http://in.mathworks.com/help/matlab/matlab_prog/vectorization.html.
[10] http://in.mathworks.com/help/matlab/ref/arrayfun.html.
[11] http://en.wikipedia.org/wiki/Pairing_function.
[12] http://in.mathworks.com/discovery/matlab-gpu.html.

GPU. GPU works with numbers (signed and unsigned integers, single-precision and double-precision floating point) only thus we convert activities across input file to distinct positive integers. We make use of `arrayfun` which is also available for GPU to do element wise operations on two large arrays. The call to `arrayfun` on GPU is massively parallelized [13]. Using `arrayfun` one call is made to parallel GPU operation that performs the entire calculation instead of making separate calls for each pair of elements. Also the data transfer overheads are incurred just once instead on each individual operation. GPU implementation is same as CPU multi-threaded implementation with the difference of one of the arguments of `arrayfun` already on GPU (with `gpuArray`) to compute direct succession on GPU. The results are brought back to CPU through `gather`.

4 Experimental Dataset

We conduct experiments on 2 real world datasets – Business Process Intelligence 2013 (BPI 2013)[13] and Business Process Intelligence 2014 (BPI 2014)[14]. BPI 2013 dataset contains logs of VINST incident and problem management system. We consider two attributes from VINST case incidents log dataset namely, Problem Number to map as Case ID and Sub-status to map as Activity. The dataset contains 13 unique activities, 7554 traces and 65533 events. BPI 2014 contains Rabobank Group ICT data. We map attribute Incident ID as the Case ID and IncidentActivity_Type as Activity in the Detail Incident Activity log. It consists of 39 unique activities, 46616 traces and 466737 events.

We create synthetic dataset due to lack of availability of very large real world data for research purposes (much larger and diverse than the BPI 2013 and BPI 2014 dataset). We first randomly define relations (causal, parallel, unrelated) between all the activities. We then use a half normal distribution with a mean and standard deviation to randomly generate the length of each trace. We create dataset A with 20 activities, standard deviation 10, mean 20 and dataset B with 50 activities, standard deviation 25, mean 50. To gain insights into the performance of parallelization strategies with different dataset sizes, each dataset is recorded for increasing trace counts. For CPU, datasets A and B are generated with trace counts 500, 2000, 8000, 32000 and 128000. Since GPUs are designed to work with large computationally intensive data [13], we generate larger dataset A and B with trace counts 10000, 50000, 250000, 1250000 and 6250000. We make our programs (MATLAB code) and synthetic data generation code publicly available[15] so that our experiments can be replicated and used for benchmarking and comparison.

5 Experimental Settings and Results

Table 1 displays the hardware and software configuration of the computer use for testing. We perform the experiments after closing the background applications

[13] doi:10.4121/500573e6-accc-4b0c-9576-aa5468b10cee.
[14] doi:10.4121/uuid:c3e5d162-0cfd-4bb0-bd82-af5268819c35.
[15] http://bit.ly/1LFJqyM.

that can affect the execution time of the MATLAB programs. We measure execution time using two MATLAB functions, namely `tic` which starts stopwatch timer and `toc` that displays the elapsed time. For all the implementations, we record time that includes both the computations involved and data transfers to and from workers or GPU. Implicit multithreading in MATLAB uses all the available cores accessible to MATLAB. We run the program using `parfor` after it is connected to specific number of workers, thus not recording time to start the parallel pool. Implicit multithreading in MATLAB uses threads equal to number of logical processor when hyperthreading[16] is enabled or uses threads equal to number of physical cores when there is no hyperthreading. The CPU that we use for performing experiments has hyperthreading enabled leading to access of 20 threads by MATLAB. `parfor` construct by default access only physical cores. Thus, in the experiments we access upto 10 workers. We calculate the speedup as S $=T_{old}/T_{new}$ where T_{old} is old execution time and T_{new} is the new execution time with improvement[17]. We set the speedup value to 1× for implementations whose execution time is considered as T_{old}.

Table 1. Machine hardware and software configuration used for experiments

Parameter	Value
CPU	Intel (R) Xeon(R) CPU E5-2670v2 @ 2.50GHz
Physical Cores	10
Logical Cores	20
Available Memory	66 GB
Operating System	Linux, 64 bit
Graphics Card	NVIDIA Tesla K40c
GPU Cores	2880
GPU Memory	12 GB
MATLAB Version	R2014b

Figures 2 and 3 shows the speedup achieved due to `parfor` and multi-threaded parallelism on Alpha Miner algorithm over CPU with T_{old} being time taken by single-threaded implementation. In Fig. 2 speedup is shown at highest trace count 128000 for dataset A and B. As shown in Fig. 2, using 2 workers good speedup values are obtained with increase in datasize, ranging from 1.55× in the smallest dataset (BPI 2013) to 6.04× in the largest dataset (dataset B). We observe with 2 workers performance improves with increase in datasize. We expect the performance to double using 4 workers but it ranges from minimum value of 2.19× (BPI 2013) to maximum of 8.60× (dataset B). Similar effect is

[16] http://www.intel.in/content/www/in/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html.

[17] http://en.wikipedia.org/wiki/Speedup.

observed with further increase in workers with speedup values increasing marginally. Marginal increase in performance happens as adding more workers leads to more communication overheads eventually reducing the gains of parallelism[18]. In fact, over the largest dataset B, performance degrades with increase in number of workers. Although due to largest size, dataset B involves maximum computations, overheads of calling workers and data transfers will also be maximum in dataset B. We observe constant drop in speedup values after adding more than 4 workers on dataset B due to large communication overheads associated with workers on it.

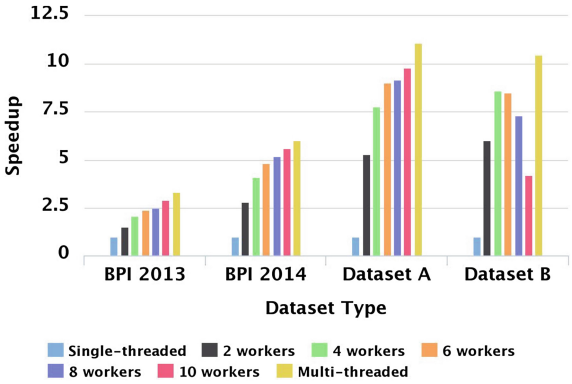

Fig. 2. Speedup gain by `parfor` and Multi-threaded Parellelism on CPU across various datasets.

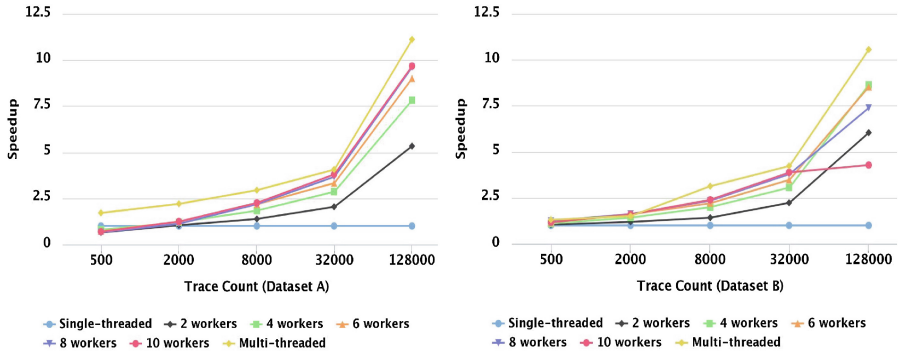

Fig. 3. Speedup gain by `parfor` and Multi-threaded Parallelism on CPU with varying dataset size.

[18] http://in.mathworks.com/company/newsletters/articles/improving-optimization-performance-with-parallel-computing.html.

Communication overheads also outweighs benefits of parallelism when computations are too less i.e. while working with smaller dataset. Figure 3 reveals that speedup value comes to be less than 1× for smallest trace count (500) in dataset A for 10 workers. In Fig. 3, speedup values increases with increase in both trace count and worker within dataset A and dataset B till the time computation time outweighs communication overheads with workers. We observe that with 20 logical cores available on machine, CPU utilisation grows approximately by 10 % with each increase in two workers on the use of `parfor`. As observed from Figs. 2 and 3, highest speedup is always achieved through multi-threaded parallelism. This can be attributed to the fact that mulithreading does not incur the cost of creating separate processes for each worker. Use of shared memory by multi-cores saves the communication and data transfer costs.

We further accelerate the algorithm on GPU after optimising it on multi-core CPU. We choose the CPU multi-threaded implementation that is implemented in the same manner as the GPU implementation for making comparisons to GPU. In Figs. 4 and 5 speedup gained by GPU is calculated with T_{old} being the exeution time of multi-threaded CPU implementation. Speedup values achieved are shown in Fig. 4 which comes out to be significant in every dataset going as far as 39.3×. Figure 5 reveals that within a given dataset with increase in trace count, GPU performance improves. The penalty of overheads associated with data transfers to and from GPU decreases with increase in datasize, hence speedup value improves with increase in trace count. Based on our experimental analysis and insights, we believe that the performance will further increase with increase in trace count. At trace count 6250000 in dataset B, the size of 'InputFile' transferred on GPU exceeds the memory limit of GPU. Thus 'InputFile' is broken into two parts and direct succession computed separately for each part. Hence we infer that GPU memory limits should be taken into account while working with GPU. The line chart in Fig. 5 reveals that performance on dataset B grows relatively slower than on dataset A. Alpha Miner on dataset B

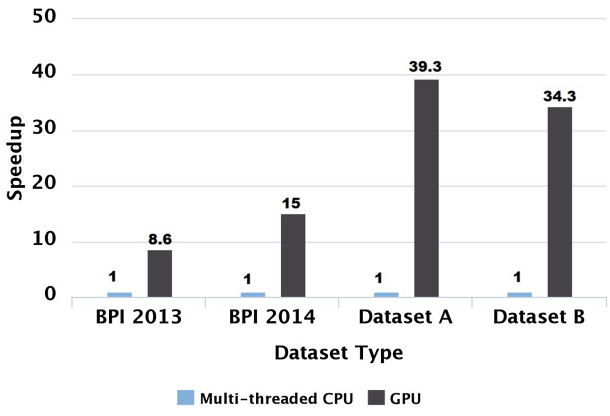

Fig. 4. Speedup gain by GPU across various datasets.

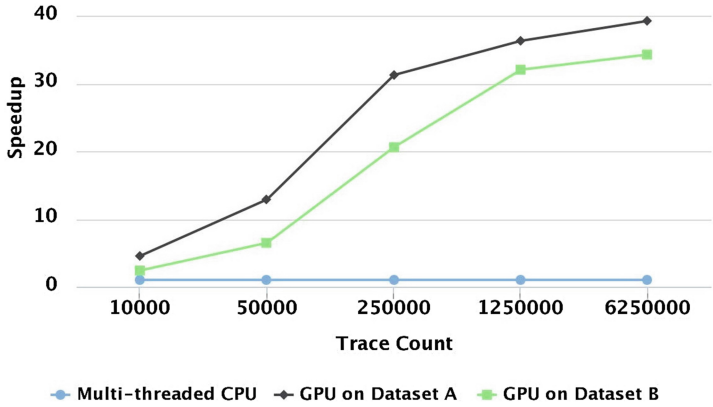

Fig. 5. Speedup gain by GPU with varying dataset size

containing larger number of activities (50) will have more of its time spend on doing activity intensive tasks like building footprint matrix, maximal set pairs generation etc. than on dataset A (20 activities). Thus on dataset A there will be larger part for doing direct succession than on dataset B. Also dataset B spends more time in transfer of data to and from GPU than dataset A due to bigger data size. Thus for every trace count point speedup on dataset B comes to be lower than on dataset A.

6 Conclusion

We conduct a series of experiments on synthetic and real world dataset which involves running computationally intensive and independent tasks of Alpha Miner algorithm in parallel on a single machine. We use MATLAB Parallel Computing Toolbox for using `parfor` to distribute computations across multiple-cores and for accessing GPU. On multi-core CPU, implicit multithreading shows higher speedup than explicit parallelism done through `parfor`. The communication overheads associated in `parfor` for creating different processes for each worker and copying data to them is significantly larger leading to reduction in parallelism benefits with addition of each worker. Alpha Miner algorithm runs with minimum execution time on GPU, showing promising speedups of as far as 39.3×.

References

1. van der Aalst, W.: Process mining: Making knowledge discovery process centric. SIGKDD Explor. Newsl. **13**(2), 45–49 (2012)
2. van der Aalst, W., Weijters, T., Maruster, L.: Workflow mining: Discovering process models from event-logs. Knowl. Data Eng. IEEE Trans. **16**(9), 1128–1142 (2004)
3. Ahmadzadeh, A., Mirzaei, R., Madani, H., Shobeiri, M., Sadeghi, M., Gavahi, M., Jafari, K., Aznaveh, M.M., Gorgin, S.: Cost-efficient implementation of k-NN algorithm on multi-core processors. In: 2014 Twelfth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE), pp. 205–208. IEEE (2014)
4. Arour, K., Belkahla, A.: Frequent pattern-growth algorithm on multi-core CPU and GPU processors. CIT **22**(3), 159–169 (2014). http://cit.srce.unizg.hr/index.php/CIT/article/view/2361
5. Cantor, G.: Ein beitrag zur mannigfaltigkeitslehre. J. fr die reine und angewandte Mathematik **84**, 242–258 (1877). http://eudml.org/doc/148353
6. Cantor, G.: Contributions to the Founding of the Theory of Transfinite Numbers. Dover, New York (1955). http://www.archive.org/details/contributionstot003626mbp
7. Desel, J., Reisig, W., Rozenberg, G. (eds.): Lectures on Concurrency and Petri Nets, Advances in Petri Nets. LNCS, vol. 3098. Springer, Heidelberg (2003). This tutorial volume originates from the 4th Advanced Course on Petri Nets, ACPN 2003, held in Eichstätt, Germany in September 2003. In addition to lectures given at ACPN 2003, additional chapters have been commissioned
8. Higham, D.J., Higham, N.J.: MATLAB Guide. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (2005)
9. Hwu, W.M.W.: GPU Computing Gems Emerald Edition, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2011)
10. Kumar, V.: Introduction to Parallel Computing, 2nd edn. Addison-Wesley Longman Publishing Co., Inc, Boston, MA, USA (2002)
11. Ligowski, L., Rudnicki, W.: An efficient implementation of Smith Waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databases. In: IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, pp. 1–8. IEEE (2009)
12. Lu, M., Tan, Y., Bai, G., Luo, Q.: High-performance short sequence alignment with GPU Acceleration. Distrib. Parallel Databases **30**(5–6), 385–399 (2012). http://dx.doi.org/10.1007/s10619-012-7099-x
13. Suh, J.W., Kim, Y.: Accelerating MATLAB with GPU Computing: A Primer with Examples, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2013)