

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

Divya Kundra

Computer Science

Indraprastha Institute of Information Technology, Delhi (IIIT-D), India

A Thesis Report submitted in partial fulfilment for the degree of

MTech Computer Science

6 June 2015

-
1. Prof. Ashish Sureka (Thesis Adviser)
 2. Prof. Saket Anand (Internal Examiner)
 3. Prof. Santonu Sarkar (External Examiner from BITS Pilani, Goa)

Day of the defense: 6 June 2015

Signature from Post-Graduate Committee (PGC) Chair:

Abstract

Process Mining consists of extracting valuable information from event logs produced by Process Aware Information Systems (PAIS) which support business processes and generate event logs as a result of execution of the supported business processes. Alpha Miner is a popular algorithm in Process Mining which consists of discovering a process model from the event logs. Discovering process models from event logs is a computationally intensive and time consuming task in context to processing large volumes of event log data. In this work, we present a parallel version of the Alpha Miner algorithm and apply different types of parallelisms (implicit, explicit, GPU) provided by MATLAB (Matrix Laboratory). To improve the program's performance, we identify its bottleneck and apply implicit parallelism on it through multithreading done by using **arrayfun** construct which perform element wise operation. For explicit parallelism, we use the **parfor** construct. We identify independent and computationally intensive **for** loops in the Alpha Miner algorithm on which **parfor** can be applied. We measure the extent of speedup achieved by implicit and explicit parallelism with respect to serial implementation of Alpha Miner algorithm on Central Processing Unit (CPU). We compare the performance obtained by implicit parallelism and explicit parallelism on CPU. Further, we use Graphics Processor Unit (GPU) to run computationally intensive parts of Alpha Miner algorithm in parallel. On GPU, we do parallelism using **arrayfun** construct. We measure the speedup achieved using GPU with respect to the same program run over multi-core CPU. Alpha Miner algorithm is accelerated the most by GPU with speedup reaching till $39.3\times$. To test the efficiency and scalability of different types of parallelisms, we perform tests on real world as well as synthetic datasets of varying sizes.

I dedicate my MTech Thesis to my family, father Mr Pradeep Kundra and mother Mrs Rajni Kundra for their endless support and love, along with my brother Mr Vikas Kundra for his immense encouragement and valuable guidance.

Acknowledgements

First and foremost I offer my deepest gratitude to my advisor, Dr. Ashish Sureka, who has supported me throughout my thesis. His patience, knowledge and confidence in me helped in working towards my goal. I owe the completion of my thesis and hence of my Masters degree to him. I could not have wished for a better advisor than him.

Besides my advisor, I would like to deeply thank my esteemed committee members Prof. Saket Anand and Prof. Santonu Sarkar for agreeing to evaluate my thesis.

My sincere thanks also goes to Prerna Juneja for helping me and spending her valuable time to review my thesis. I would also like to thank IIITD IT and computing infrastructure team for granting access to the machines used for carrying experiments. Last but not the least, I would like to thank all my family members who encouraged and kept me motivated throughout the thesis.

Declaration

This is to certify that the MTech Thesis Report titled **Vidushi: Parallel Implementation of Alpha Miner Algorithm and Performance Analysis on CPU and GPU Architecture** submitted by **Divya Kundra** for the partial fulfillment of the requirements for the degree of *MTech* in *Computer Science* is a record of the bonafide work carried out by her under my guidance and supervision at Indraprastha Institute of Information Technology, Delhi. This work has not been submitted anywhere else for the reward of any other degree.

Professor Ashish Sureka

Indraprastha Institute of Information Technology, New Delhi

Contents

List of Figures	vi
List of Tables	viii
1 Research Motivation and Aim	1
1.1 Process Mining	1
1.1.1 α Miner Algorithm	4
1.2 Parallel Computing	7
1.2.1 MATLAB	8
1.3 Problem Motivation, Definition and Aim	9
2 Related Work and Research Contributions	11
2.1 Related Work	11
2.1.1 Process Mining	11
2.1.2 Multi-Core CPU for Data Mining Algorithms	12
2.1.3 Multi-Core GPU for Data Mining Algorithms	12
2.2 Novel Research Contributions	13
3 Research Framework and Solution Approach	14
3.1 Sequential Single Threading on CPU	16
3.2 Explicit Parallelism on CPU	18
3.3 Multithreading Parallelism in CPU	22
3.4 Graphics Processing Unit (GPU)	24
4 Experimental Dataset	27
5 Experimental Settings and Results	33

CONTENTS

6	Limitations and Future Work	42
7	Conclusion	43
	References	44

List of Figures

1.1	Types of Process Mining [1]	2
1.2	A Petri Net consisting of places and transitions.	3
3.1	Parallelization over determining Direct Succession Relation.	15
3.2	Parallelization over building Footprint Matrix.	16
3.3	Parallelization over forming Set Pairs (A,B).	16
3.4	Fragments of Alpha Miner Algorithm Implementation in MATLAB Code showing programming constructs for Multi-Core CPU and GPU implementations.	17
(a)	Single-threaded implementation.	17
(b)	parfor implementation.	17
(c)	Multi-threaded implementation.	17
(d)	GPU implementation.	17
3.5	Process creation for each of the worker in parfor while using 4 workers.	19
3.7	arrayfun mechanism	22
3.8	Computational Resources of CPU and GPU [2].	25
4.1	Dataset BPI 2013.	28
4.2	Case Duration for BPI 2013.	28
4.3	Dataset BPI 2014.	29
4.4	Case Duration for BPI 2014.	29
4.5	Synthetic Dataset A.	31
4.6	Synthetic Dataset B.	31
5.1	Speedup gain by parfor and Multi-threaded Parallelism on CPU across various datasets.	35

LIST OF FIGURES

5.2	Speedup gain by parfor and Multi-threaded Parallelism on CPU with varying dataset size.	36
5.3	CPU Utilisation with varying workers.	37
5.4	Performance of different parfor loops across datasets.	38
5.5	Speedup gain by GPU across various datasets.	39
5.6	Speedup gain by GPU with varying dataset size.	40
5.7	Time taken to perform all GPU related tasks (data transfer, gather and arrayfun) on Dataset A and Dataset B.	40

List of Tables

1.1	Event Log.	4
1.2	Sequential Event Log.	4
1.3	Direct Succession Relation.	5
1.4	Causal Relation.	5
1.5	Parallel Relation.	5
1.6	Unrelated Relation.	5
1.7	Footprint Matrix.	7
1.8	Set Pair (A,B)	7
1.9	Maximal Set Pair (A,B).	7
4.1	Details of Experimental Datasets.	32
5.1	Machine Hardware and Software Configuration used for Experiments . .	33
5.2	Execution Time (sec) of Single-threaded, <code>parfor</code> and Multi-threaded implementations of Alpha Miner Algorithm on CPU.	34
5.3	Execution Time (sec) of Multi-threaded CPU and GPU implementations of Alpha Miner Algorithm.	38

1

Research Motivation and Aim

1.1 Process Mining

Process mining is extraction of valuable insights from event logs produced by Process Aware Information System (PAIS). PAIS are software systems managing and executing operational processes that involves people, applications and information sources on the basis of process models within an organisation [3]. Examples of PAIS systems includes Workflow Management Systems (WMS) and Business Process Management Systems (BPMS) [3]. An event log stores the detailed information about each development represented as an event which is executed in the process. An event is stored as the combination of 4 fields: activity (well defined step in a process), Case Id, (identifier for which activity is recorded), actor (person starting and performing the activity) and timestamp (the beginning time of the event) [1]. A trace/case/process-instance is defined as the sequential representation of all the activities belonging to the same Case Id in order of their occurrence with respect to time. A process model is a visual representation of the process which is recorded by event log. The event logs produced by PAIS systems are used to produce process models as shown in Figure 1.1. Process models gives clear-cut business insights by presenting a directed graph in terms of nodes and edges. Nodes represent the activities that are performed in the process while edges connect the activities in order of their occurrence with respect to time. If an activity A has a earlier timestamp than activity B, then there will be an edge from node representing activity A to node representing activity B. There are several notations to represent the process models such as petri nets [4], causal nets [5] etc. A petri net [4] is

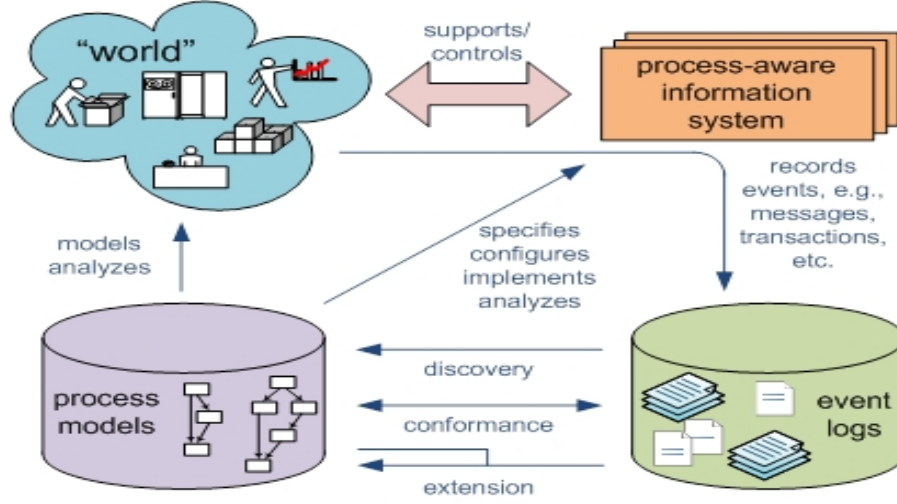


Figure 1.1: Types of Process Mining [1]

also known as place/transition net. While a transition represents the activity executed in the process, place denotes the pre or post conditions that must be followed to reach from one transition to another. Petri net is represented by 3 tuple value (S, T, W) where S and T are finite sets of places and transitions respectively. W is a multi set of arcs. One arc can connect one place and transition only. As seen from Figure 1.2, places are represented with circles, transitions are activities represented inside rectangles and arcs are denoted by directed arrows that show connection between places and transitions or vice-versa. In our research we implement the Alpha Miner algorithm that gives Petri net model as an output. Following are the three perspectives of process mining [6]:

1. Process Perspective [6]: This perspective aims at producing process models from the event log focusing on order of the occurrence of events. It is also known as Control flow perspective. The aim of this perspective is to find good characterisation of all the possible paths that exist between the nodes representing activities.
2. Organisational Perspective [6]: This perspective focuses on determining the structure of the organisation from people involved, their roles and relationships. It focuses on the actor field of the event log.
3. Case Perspective [6]: This perspective focuses on properties of cases. It aims

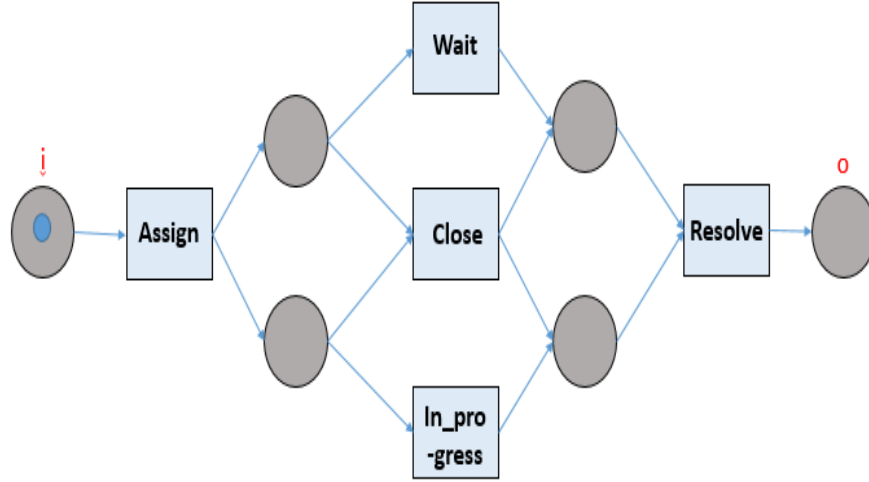


Figure 1.2: A Petri Net consisting of places and transitions.

at analysing properties of cases like the count of involved activities, represented Case Id for particular set of activities etc.

As shown in Figure 1.1, process mining can be of the following types [1]:

1. Discovery [1]: It takes an event log as an input to produce process model for the recovery of hidden and valuable information.
2. Conformance [1]: It checks if the process model and event log have conformance with each other. It helps in monitoring all the deviations that occur in organisation's process.
3. Enhancement [1]: It aims at improvement of the existing process. Generated process models may indicate the need for improving the current standard of the workflow.

Some of the advantages¹ of process mining are understanding and identifying the root causes behind problems, locating anomalies, finding bottlenecks by detecting the regions (nodes or edges) where the largest delays occur, achieving compliance by quantifying the amount by which the process models deviate from real-life, validating process changes and understanding the entire process.

¹<http://www.bptrends.com/the-added-value-of-process-mining/>

Table 1.1: Event Log.

Case Id	Activity
1	Assign
1	Wait
1	In_progress
1	Resolve
2	Assign
2	Close
2	Resolve
3	Assign
3	In_progress
3	Wait
3	Resolve

Table 1.2: Sequential Event Log.

Activity
Assign, Wait, In_progress, Resolve
Assign, Close, Resolve
Assign, In_progress, Wait, Resolve

1.1.1 α Miner Algorithm

α Miner is one of the most popular process mining algorithm that was proposed by Wil van der Aalst, Ton Weijters, and Laura Maruster in 2004. It is a fundamental process discovery algorithm that extracts a process model from an event log consisting of traces and represents it as a petri net [7]. It is based on analyzing immediate successor relation between activities present in the event log. Table 1.1 shows an event log which is converted to a sequential event log L as shown in Table 1.2 in which the activities belonging to the same Case Id appear sequentially in increasing order of timestamp. The algorithm scans through a sequential event log like one shown in Figure 1.2 and establishes the following relations [7] between all the activities:

1. Direct Succession ($a \succ_L b$) if activity a directly precedes activity b in some trace i.e. activity a having earlier timestamp value is immediately followed by activity b which is performed just after a .
2. Causal ($a \rightarrow_L b$) if $a \succ_L b$ and $b \not\prec_L a$.
3. Parallel ($a \parallel_L b$) if $a \succ_L b$ and $b \succ_L a$.
4. Unrelated ($a \#_L b$) if $a \not\prec_L b$ and $b \not\prec_L a$.

For the sequential event log L shown in Table 1.2 the pair of activities having direct succession is shown in Table 1.3. In Table 1.4 causal relation computed from direct

Table 1.3: Direct Succession Relation.

Activity <i>a</i>	Activity <i>b</i>
Assign	Wait
Wait	In_progress
In_progress	Resolve
Assign	Close
Close	Resolve
Assign	In_progress
In_progress	Wait
Wait	Resolve

Table 1.4: Causal Relation.

Activity <i>a</i>	Activity <i>b</i>
Assign	Wait
Assign	In_progress
Assign	Close
Wait	Resolve
In_progress	Resolve
Close	Resolve

Table 1.5: Parallel Relation.

Activity <i>a</i>	Activity <i>b</i>
Wait	In_progress
In_progress	Wait

Table 1.6: Unrelated Relation.

Activity <i>a</i>	Activity <i>b</i>
Assign	Resolve
Wait	Close
In_progress	Close
Resolve	Assign
Close	Wait
Close	In_progress
Assign	Assign
Wait	Wait
In_progress	In_progress
Resolve	Resolve
Close	Close

succession for event log *L* is displayed. Table 1.5 represents the parallel relation for event log *L* which is also computed with the help of direct succession. Table 1.6 reports the unrelated relation for event log *L* computed with the help of direct succession.

Let the event log *L* shown in Table 1.2 be over *T* activities or transitions. The detailed steps involved in the Alpha Miner algorithm are as follows:

1. $T_L = \{ t \in T \mid \exists \sigma \in L \ t \in \sigma \}$

Determine the set of unique activities present in the event log. In event log *L*, T_L from *T* is {Assign, Wait, In_progress, Resolve and Close}.

2. $T_I = \{ t \in T \mid \exists \sigma \in L \ t = \text{first}(\sigma) \}$

From the set of all activities *T*, determine the set of activities which do not have

immediate predecessor anywhere in any of the trace in the log. In event log L, T_I is {Assign}.

3. $T_O = \{ t \in T \mid \exists \sigma \in L \ t = \text{last}(\sigma) \}$

From the set of all activities T, determine the set of activities which do not have immediate successor anywhere in any of the trace in the log. In event log L, T_O is {Resolve}.

4. Scan through the traces present in the sequential event log and determine the above mentioned relations (\succ , \rightarrow , \parallel , \sharp) between all activities and represent them in the form of a matrix called footprint. Table 1.7 shows footprint matrix for L.
5. $X_L = \{ (A,B) \mid A \subseteq T_L \wedge A \neq \emptyset \wedge B \subseteq T_L \wedge B \neq \emptyset \wedge \forall a \in A \ \forall b \in B \ a \rightarrow_L b \wedge \forall a1, a2 \in A \ a1 \sharp_L a2 \wedge \forall b1, b2 \in B \ b1 \sharp_L b2 \}$

Using the footprint matrix we generate X_L that consists all possible pairs of sets (A,B) such that activities within set A and within set B are unrelated to rest of the activities of their set and each activity in set A is in causal relation with every activity of set B. Table 1.8 reports X_L for L.

6. $Y_L = \{ (A,B) \in X_L \mid \forall (A', B') \in X_L \ A \subseteq A' \wedge B \subseteq B' \implies (A,B) = (A', B') \}$

In X_L , if for a set pair (A,B), all activities in A are a subset of activities in set A' and all activities in set B are a subset of activities in set B' and (A', B') set pair is present in X_L , then set pair (A,B) in X_L is considered to be same as set (A', B') . Table reports 1.9 Y_L for L.

7. $P_L = \{ p_{(A,B)} \mid (A,B) \in Y_L \} \cup \{ i_L, o_L \}$

In P_L , a place (discussed in Section 1.1) is generated for each distinct pair of set (A,B). Along with it, a input place i_L and output place o_L are generated.

8. $F_L = \{ (a, p_{(A,B)}) \mid (A,B) \in Y_L \wedge a \in A \} \cup \{ (p_{(A,B)}, b) \mid (A,B) \in Y_L \wedge b \in B \} \cup \{ (i_L, t) \mid t \in T_I \} \cup \{ (t, o_L) \mid t \in T_O \}$

For each set pair (A,B) in Y_L , arcs are connected from every activity present in set A to a place generated for the set pair (A,B) and arcs are also connected from the place to every activity present in set B. Activities in T_I are connected to the input place i_L and activities in T_O are connected to the output place o_L .

9. $\alpha(L) = (P_L, T_L, F_L)$

The generated petri net of Alpha Miner algorithm is represented by P_L , T_L and F_L as shown in Figure 1.2.

Table 1.7: Footprint Matrix.

	Assign	Wait	In_progress	Resolve	Close
Assign	#	→	→	#	→
Wait	←	#		→	#
In_progress	←		#	→	#
Resolve	#	←	←	#	←
Close	←	#	#	→	#

Table 1.8: Set Pair (A,B)

Set A	Set B
Assign	Wait
Assign	In_progress
Assign	Close
Wait	Resolve
In_progress	Resolve
Assign	{Wait, Close}
Assign	{In_progress, Close}
Close	Resolve
{Wait, Close}	Resolve
{In_progress, Close }	Resolve

Table 1.9: Maximal Set Pair (A,B).

Set A	Set B
Assign	{Wait, Close}
Assign	{In_progress, Close}
{Wait, Close}	Resolve
{In_progress, Close }	Resolve

1.2 Parallel Computing

Since the advent of multiprocessors system, parallel programming languages have been designed to make full use of parallelism. Out of many available languages, MPI¹ and OpenMP² are the ones that are widely used and accepted³. While MPI is a distributed memory model that works on distributed computers, OpenMP uses shared memory

¹<http://www.mcs.anl.gov/research/projects/mpi/>

²<http://openmp.org/wp/about-openmp/>

³https://vlebb.leeds.ac.uk/bbcswebdav/orgs/SCH_Computing/FYProj/reports/1213/Hussain.pdf

model and works on multiprocessors [8]. MPI uses messages to communicate while OpenMP is directive based. MPI is difficult to debug and communication overheads can affect program's performance while performance of OpenMP is limited by thread management and cache coherency. CUDA¹ (Compute Unified Device Architecture) is a breakthrough language in parallel processing on GPU. The capabilities of GPU have expanded and there is a remarkable performance gain that is observed [9]. In paper *Scalable parallel programming with CUDA* [10], the author has investigated how CUDA is a revolution in parallelisation and can be adapted to accelerate any general purpose application. Java also provides multithreading capabilities with the performance and scalability tested on multi-core systems [11]. Python too has libraries to use multicore CPUs or multiple CPUs. Tools such as Parallel Virtual Machine², allows multiple heterogeneous computers to be used as one parallel distributed computer.

1.2.1 MATLAB

MATLAB³ is a high level programming language used for various scientific and engineering calculations that is developed by MathWorks. It provides interactive environment for problem exploration and design, offers various mathematical functions and development tools for improving code and features for integrating program with programs written in other languages. It provides 3 different ways to do parallelism⁴:

1. **Multi-threaded Parallelism:** Some of MATLAB's inbuilt functions⁵ implicitly provide multithreading. A single MATLAB process generates multiple instruction streams. CPU cores execute these streams while sharing the same memory.
2. **Explicit Parallelism:** In this type of parallelism multiple instances of MATLAB run on separate processors often each with its own memory and simultaneously execute the code that externally invokes these instances.

¹http://www.nvidia.com/object/cuda_home_new.html

²<http://www.csm.ornl.gov/pvm/>

³<http://in.mathworks.com/products/matlab/?refresh=true>

⁴<http://in.mathworks.com/company/newsletters/articles/parallel-matlab-multiple-processors-and-multiple-cores.html>

⁵<http://www.mathworks.com/matlabcentral/answers/95958-which-matlab-functions-benefit-from-multithreaded-computation>

- 3. Distributed Computing:** Multiple instances of MATLAB run independent computations on each computer, each with its own memory.

Distributed Computing is an extension of explicit parallelism and can be easily extended to explicit parallelism using MATLAB Distributed Computing Server¹ toolbox. Multithreading is performed using inbuilt multi-threaded functions. Use of Parallel Computing Toolbox² is done to make use of different cores of CPU and also to access the GPU. MATLAB provides the analysis of code performance through the Code Analyser which checks the code while it is being written suggesting ways to improve performance and through the Profiler which shows how much time is taken by each line of code. We make use of both of them in determining the most computational intensive tasks and in optimising them.

1.3 Problem Motivation, Definition 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, for the purpose of process improvement and enhancement. 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 log data generated by PAIS. Organisation whose processes are recorded and monitored by IT Systems have increased tremendously. More people are involved in these organisations leading to occurrence of more events and adding more data to the event logs. It becomes computationally intensive and time consuming for process discovery algorithms to work on ever increasing large sized event logs. Process discovery algorithms gives a clear cut insights of business, helping in enhancing the current workflow practices of the organisations. To improve the current standards of workflow, organisations make use of process discovery algorithm. Thus there is a need to make the process discovery algorithms efficient enough to handle the rapidly growing size of event logs. Distributed and Grid computing, parallel execution on multi-core processors and using hardware accelerators such as Graphics Processing Unit (GPU)

¹<http://in.mathworks.com/products/distriben/>

²<http://in.mathworks.com/products/parallel-computing/>

1.3 Problem Motivation, Definition and Aim

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 consisting of discovering a process model from event logs. Our analysis of the Alpha Miner algorithm reveals that the algorithm contains independent tasks which can be split among different processors or threads and the algorithm has the ability or property of parallelization. The work presented in this research is motivated by the need to investigate the efficiency and scalability of Alpha Miner algorithm on multi-core processors (multi-core parallel processing) and GPU based hardware accelerators.

We propose a parallel approach for Alpha Miner algorithm. We test the approach by applying the implicit multi-threaded Parallelism and Explicit Parallelism on Alpha Miner over multi-core CPU. Along with this, we also take help of GPU to perform parallelization. The objective is to accelerate Alpha Miner with the help of different parallelisms provided by MATLAB. We determine the parallelism model that is best suitable for the problem under consideration.

2

Related Work and Research Contributions

In this chapter we discuss previous work that is closely related to our research and list the novel research contributions of our work in context to already existing work.

2.1 Related Work

The related work has been categorised into three lines of research.

2.1.1 Process Mining

Several process discovery algorithms have been proposed in literature of process mining. Aalst et al. present Alpha Miner algorithm, the most popular and fundamental algorithm to discover petri nets from event logs [7]. In our research we apply parallelization on Alpha Miner algorithm. To mine the unstructured real life event logs, Aalst et al. propose the Fuzzy Miner algorithm [12]. The algorithm uses abstraction and clustering to discover models that are easy to comprehend [12]. Weijters et al. propose Heuristics Miner [13], an algorithm which deals with noise and presents the main behavior of the event log. The algorithm includes constructing the dependency graph, constructing the input output expression for each activity in the graph and search for longest distance dependency relation [13]. Dongen et al. propose Multi-phase Miner [14][15] that uses OR split/join semantics. The authors have proposed an approach that builds the instance graph based on the information present in the event log. The

results are represented as Event-driven Process Chains (EPCs). It expresses complex behavior in relatively well structured models [14][15].

2.1.2 Multi-Core CPU for Data Mining Algorithms

Implementation of data mining algorithms on multi-core CPU is an area that has attracted several researchers attention. Ahmadzadeh et al. [16] 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 times. Matsumoto et al. [17] propose an improved parallel algorithm for outlier detection on uncertain data using density sampling and develop an implementation running on both GPUs and multi-core CPUs. Nan et al. in [18] implement parallel version of Global and Coarse-Grained genetic algorithms using MATLAB Parallel Computing Toolbox and Distributed Computing Server software and achieve a higher speedup and better performance. Stratton et al. in [19] propose a new framework MCUDA that allows CUDA programs to be executed on shared memory and multi-core CPUs proving that CUDA can be used for architectures other than GPU. Hong et al. in [20] present a parallel implementation of the popular Breadth First Search algorithm on multi-core CPU and also study the effects of the proposed architecture on BFS performance by comparing multiple CPU and GPU systems as well as a quad-socket high-end CPU system.

2.1.3 Multi-Core GPU for Data Mining Algorithms

GPU has brought a big revolution in parallelising the algorithms. Implementation of data mining algorithms on multi-core GPU is an area that also has attracted several researchers attention. Ligowski et al. in [21] uses CUDA programming environment on Nvidia GPUs and ATI Stream Computing environment on ATI GPUs to speedup the popular Smith Waterman sequence alignment algorithm. Their implementation strategy achieves a 3.5 times higher per core performance than the previous implementations of this algorithm on GPU. Arour et al. [22] present two FP-growth (Frequent Pattern) implementations that take advantage of multi-core processors and utilize new generation GPUs. Stuart et al. in [23] present a MapReduce library that uses the power of GPU clusters for large-scale computing and compares this library with a highly-optimized multi-core MapReduce and another GPU-MapReduce library to show the

power of the proposed library. Lu et al. [24] develop a method which adopts the GPU as a hardware accelerator to speedup the sequence alignment process. Paula et al. in [25] uses the CUDA-Matlab integration to parallelise the hybrid BiCGStab (bi-conjugate gradient stabilized) iterative method in a Graphics Processing Unit and achieves a speedup of $76\times$ and $6\times$ when compared to implementations in C language and CUDA-C integration.

2.2 Novel Research Contributions

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 its execution performance by using multi-core CPU.
2. A focused study on accelerating Alpha Miner algorithm through parallelism on the GPU and testing the approach on various real and synthetic datasets.

3

Research Framework and Solution Approach

Alpha Miner is one of the most popular process discovery algorithm in the field of process mining. To parallelize Alpha Miner algorithm we identify discrete and independent tasks in it which can be solved concurrently. MATLAB supports parallelization only for independent tasks. Thus through manual analysis of the algorithm we determine which of the steps in Alpha Miner algorithm can be parallelized. As explained in Section 1.1.1 the first step is finding the set of unique activities present in the sequential event log. For this step, if different threads are allowed to operate on different set of activities, they would require communication among them to select the set of unique activities. Thus Step 1 cannot be broken into of independent tasks. To reduce the execution time of the overall algorithm, outputs T_I and T_O for Step 2 and Step 3 respectively can be find while determining the causal relation during building of the footprint matrix instead of examining the entire event log. Since we parallelize building the footprint matrix, Step 2 and Step 3 are in-turn parallelized. Parallelization can very well be applied while determining relations (direct succession, causal, parallel and unrelated) in Step 4. As shown in Figure 3.1 for sequential event log L shown in Table 1.2 parallelization can be applied on traces, different threads can work on different traces and determine the pair of activities having direct succession relation. Results can be gathered from all the threads and after removing redundancies we can get unique pair of activities having direct succession relation. Parallelization can again be applied to determine rest of the relations (causal, parallel, unrelated) for all the activities, and footprint matrix can

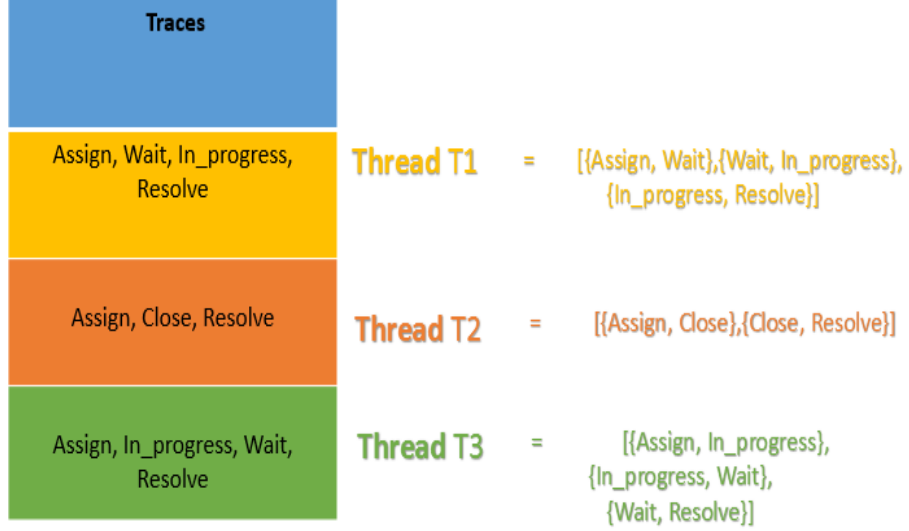


Figure 3.1: Parallelization over determining Direct Succession Relation.

thus be build. As shown in Figure 3.2 for log L, different threads can work on different activities and find the relations of those activities with the rest of the activities present in the log. Different rows of the footprint can thus simultaneously be worked by different threads. Step 5 also involves independent tasks. As displayed in Figure 3.3 for log L, a thread can determine all the set pairs (A,B) that can be formed by including a particular activity in set A. Similarly, other threads can simultaneously work on different activities and find out all the pair of sets (A,B) that can be generated by keeping a activity in set A. Results can be collected from all the threads. While gathering the results, redundancies can be removed and only maximal set pairs as required in Step 6 can be chosen. Step 7, Step 8 and Step 9 involves forming the petri net graph from the maximal set pairs. Again parallelization can be make use of by different threads connecting arcs to and from places and transitions for a pair of sets (A,B). We have not produced the petri net graph in MATLAB, thus have not make use of parallelizations in building the petri net. We see that except for determining unique activities and producing the graph, we make use of parallelization in rest of the steps. Thus for Alpha Miner algorithm discussed in Section 1.1.1 we make use of parallelization on its 5 out of 9 steps. We first implement Alpha Miner in serial mode then we implement three different types of parallelisms on it. For all the implementations, we encode the activity names by unique integers instead of using their string representation.

3.1 Sequential Single Threading on CPU

	Assign	Wait	In_progress	Resolve	Close	
Assign	#	→	→	#	→	Thread T1
Wait	←	#		→	#	Thread T2
In_progress	←		#	→	#	Thread T3
Resolve	#	←	←	#	←	Thread T4
Close	←	#	#	→	#	Thread T5

Figure 3.2: Parallelization over building Footprint Matrix.

Activities	
Assign	Thread T1 = [{Assign, Wait},{Assign, In_progress},{Assign, Close}, {Assign, {Wait, Close}},{Assign, {In_progress, Close}}]
Wait	Thread T2 = [{Wait, Resolve},{Wait, Close},Resolve]
In_progress	Thread T3 = [{In_progress, Resolve},{In_progress, Close},Resolve]
Resolve	Thread T4 = [Empty]
Close	Thread T5 = [{Close, Resolve}, {Wait, Close},Resolve], {In_progress, Close},Resolve]

Figure 3.3: Parallelization over forming Set Pairs (A,B).

3.1 Sequential Single Threading on CPU

In sequential programming, there is an ordered relationship of execution of instructions where only a single instruction executes at a particular instance of time¹. The program is executed over a single processor only. Serial implementation done on a single thread provides a base for evaluating comparisons from other implementations

¹https://computing.llnl.gov/tutorials/parallel_omp/

3.1 Sequential Single Threading on CPU

(multi-threaded, **parfor**). To achieve serial version of Alpha Miner algorithm, we ensure not to use any inbuilt multi-threaded functions¹ available in MATLAB. We enable only a single thread in the program by using **maxNumCompThreads(1)**. Use of **for** loop throughout the program makes it sequential in nature. As shown in Figure 3.4(a) we have implemented the main functionalities in Alpha Miner algorithm like determining all the direct succession relations by scanning the entire log, building the footprint with its help and determining the maximal set pairs through **for** loop. Use of **for** loops makes the program slower as at each iteration conditions are checked and branching occurs adding to more overheads and affecting the code's performance.

```
for i=1:traces
%Find Direct Succession
end
```

```
for i=1:activities
%Build Footprint Matrix
end
```

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

(a) Single-threaded
implementation.

```
parfor i=1:traces
%Find Direct Succession
end
```

```
parfor i=1:activities
%Build Footprint Matrix
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.

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

(d) GPU implementation.

Figure 3.4: Fragments of Alpha Miner Algorithm Implementation in MATLAB Code showing programming constructs for Multi-Core CPU and GPU implementations.

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

3.2 Explicit Parallelism on CPU

In contrast to sequential processing, parallel processing lets execution of multiple tasks at the same time [26]. In parallel processing, the instructions are distributed to different processors which work simultaneously in order to complete the task fast. The ease and success of parallelism depends on how much synchronisation exists between the divided tasks. Speedup will be maximum when tasks are independent i.e. there is no communication between tasks executing in parallel [27]. Parallel computing is done on multi-core computers.

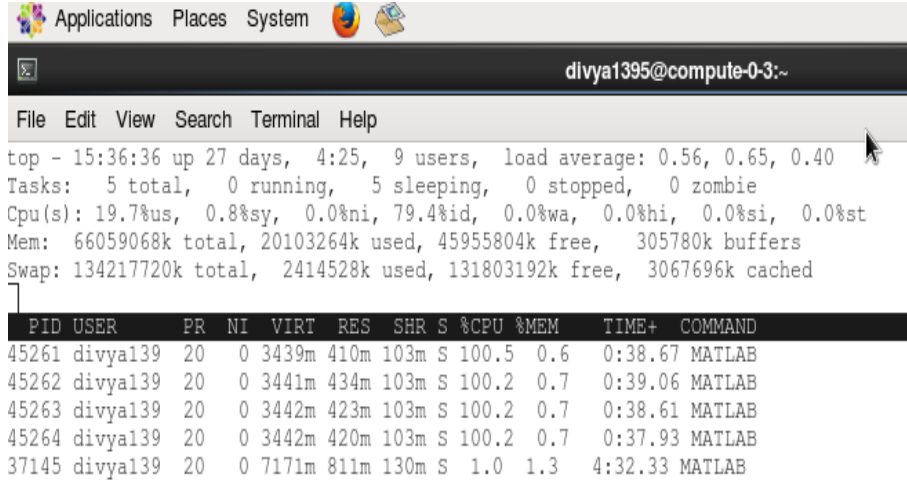
MATLAB has the provision of applying external parallelism over set of independent tasks through `parfor`¹. `parfor` allows execution of the loop iterations in parallel on labs. Labs are workers which are executed on processor cores. Syntax of `parfor` is shown by Equation (3.1). 'loopvar' is the variable for which iterations occurs, 'initval', 'endval' are limits of the loop which must be finite integers and 'statements' represent the code required to be parallel.

$$\text{parfor loopvar} = \text{initval}:\text{endval}, \text{statements}, \text{end} \quad (3.1)$$

`parfor` mechanism can be summarised as follows:

1. Using `parfor` separate process is created for each worker having its own memory and own CPU usage. As shown in Figure 3.5, using 4 workers creates 4 different MATLAB processes. They are headed by a client process which manages them.
2. When `parfor` is executed, the MATLAB client communicates with the MATLAB workers which form a parallel pool.
3. The code within the `parfor` loop is distributed to workers and it executes in parallel in the pool.
4. The required data needed by workers to do the computations is send from client to all the workers and the results from all the workers are collected back by client as shown in Figure 3.6.

¹<http://in.mathworks.com/help/distcomp/parfor.html>



```

top - 15:36:36 up 27 days, 4:25, 9 users, load average: 0.56, 0.65, 0.40
Tasks:  5 total,  0 running,  5 sleeping,  0 stopped,  0 zombie
Cpu(s): 19.7%us,  0.8%sy,  0.0%ni, 79.4%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem: 66059068k total, 20103264k used, 45955804k free, 305780k buffers
Swap: 134217720k total, 2414528k used, 131803192k free, 3067696k cached

  PID USER      PR  NI  VIRT  RES  SHR S  %CPU  %MEM    TIME+  COMMAND
 45261 divya139   20   0 3439m 410m 103m S 100.5   0.6   0:38.67  MATLAB
 45262 divya139   20   0 3441m 434m 103m S 100.2   0.7   0:39.06  MATLAB
 45263 divya139   20   0 3442m 423m 103m S 100.2   0.7   0:38.61  MATLAB
 45264 divya139   20   0 3442m 420m 103m S 100.2   0.7   0:37.93  MATLAB
 37145 divya139   20   0 7171m 811m 130m S   1.0   1.3   4:32.33  MATLAB

```

Figure 3.5: Process creation for each of the worker in `parfor` while using 4 workers.

5. 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 there is dependency between different loop iterations, an error will be produced on using `parfor` and the dependency is required to be removed in order to proceed.
6. 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. Thus `parfor` distributes loop iterations in chunks to be executed by the worker.
7. `parfor` can be useful in situations where there are many iterations of simple calculation loop that can distributed to a large number of workers so that each workers completes some portion of the total iterations. It can also be used in cases where loop iteration take long time to execute so that by simultaneous execution by multiple workers it can be completed faster.
8. To start a pool of workers Equation (3.2) `parpool` 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.

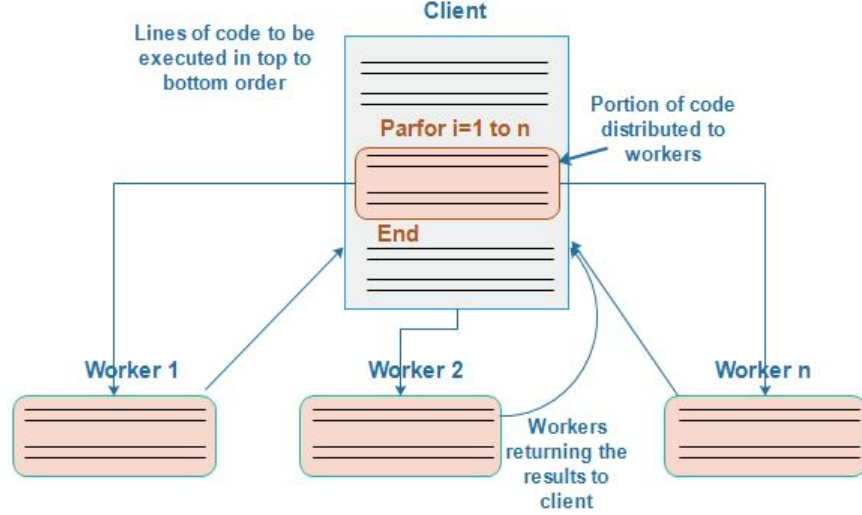


Figure 3.6: Portion of code executing in parallel using the `parfor` language construct¹

$$\text{parpool}(\text{profilename}, \text{poolsize}) \quad (3.2)$$

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 enabling us to apply `parfor`:

- (a) **Determining direct succession relation:** The causal, parallel and unrelated relations are in-turn derived from the direct succession relation. Discovering the pair of direct succession activities is an independent task and can be distributed to different workers for e.g. first worker can calculate the direct succession relations from one trace, second worker from some other trace and so on. As shown in Figure 3.4(b), first `parfor` loop distributes the total 'traces' present in input file among various workers. The results can be gathered from each thread, redundancies can be removed and unique pair of activities having direct succession between them can be deduced.
- (b) **Building up the footprint:** The task of creating the footprint i.e. finding the relations of all the activities with one another can be broken into

¹<http://in.mathworks.com/help/stats/working-with-parfor.html>

independent work of finding all the relations a activity (causal, parallel, unrelated) with the rest of the activities separately by a worker. In the second **parfor** loop of Figure 3.4(b), each worker upon receiving a activity out of total unique 'activities' computes the relations of that activity with the remaining activities present in event log.

- (c) **Forming maximal set pairs:** The process of forming the maximal set pairs, can also be broken into smaller unrelated processes 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 Figure 3.4(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 retain only the maximal pair sets from the gathered results 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. Direct succession relation is computed by scanning the event log thus its computational time is dependent on count and length of traces present in the event log. Computational time for other relations (causal, parallel, unrelated) which are determined through direct succession depend on count of activities, as more are the activities, more will be the entries in footprint matrix. Time for calculating maximal set pair is also dependent on total activities present in the log. More are the activities, more time will be spend in generating the sets. 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 number of traces to be scanned for determining direct succession by order of thousands 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.

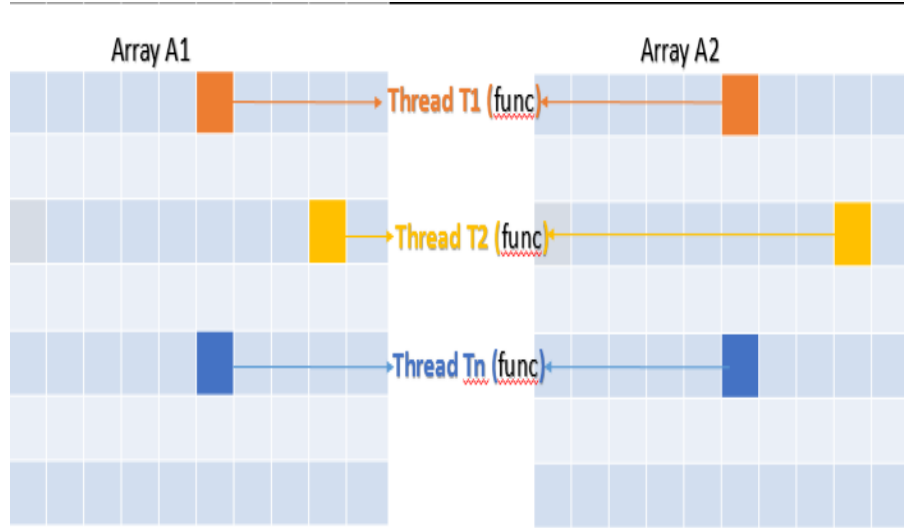


Figure 3.7: arrayfun mechanism

3.3 Multithreading Parallelism in CPU

In MATLAB by default implicit multithreading¹ is provided for expressions and functions that are combinations of element wise operations. In this type of parallelism, multiple instruction streams are generated by one instance of MATLAB session². Multiple cores share the memory of a single computer to access these streams. The 3 basic requirements to achieve it are:

- (a) Each element wise operation should be independent of each other.
- (b) Size of data should be big enough so that speed up achieved by the concurrent execution exceeds the time required for partitioning and managing different threads.
- (c) The function in consideration should preferably be complex and challenging enough so that higher speedups are achieved while multithreading.

To fulfil these requirements, independent and big tasks vectorization³ are essential for implicit parallel computations. Vectorization is one of the most efficient ways

¹<http://www.mathworks.com/matlabcentral/answers/95958-which-matlab-functions-benefit-from-multi-threaded-computation>

²<http://in.mathworks.com/company/newsletters/articles/parallel-matlab-multiple-processors-and-multiple-cores.html>

³http://in.mathworks.com/help/matlab/matlab_prog/vectorization.html

of writing the codes in MATLAB [28]. It performs operations on large matrices through a single command at once instead of performing each operation one by one inside the **for** loop. An effective way of using multithreading 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 [28].

Through experiments we determine bottleneck in finding the direct succession relations. The determination of direct succession can be vectorised using **arrayfun**¹. MATLAB uses implicit multithreading using commands such as- **arrayfun**. Syntax is as shown by Equation (3.3). **arrayfun** applies the function specified in function handle 'func' to each element of equal sized input arrays A1,..An. Shown in Figure 3.7, is the working mechanism of **arrayfun** construct. In Figure 3.7 there are two equal sized input arrays A1 and A2. Independent threads work simultaneously on corresponding elements of these two arrays and each thread apply the function 'func' passed through **arrayfun** to the two elements of the arrays. The order of execution of threads is not specific, thus element wise operation should be independent of each other. The implementation of **arrayfun** in the algorithm is as represented in Figure 3.4(c). 'InputFile' is the sequential event log input to the program. 'ShiftedFile' is obtained by shifting the 'InputFile' by 1 to the left. Each cell of the 'ShiftedFile' contains the immediate succeeding activity of the activity present in the corresponding cell of 'InputFile'. Since 'func' operates over each two corresponding elements of the input arrays, through 'func' we get the pair of activities a and b such that $a \succ_L b$ i.e. they hold direct succession relation. We apply Cantor pairing function [29] [30] [31] in 'func' by which each activity a in the input file and its immediate successor b in the left shifted input file are uniquely encoded. Syntax of Cantor pairing function is given in Equation 3.4 where a and b are activities having direct succession relation $a \succ_L b$. Since we have encoded all activities as unique integers, Cantor pairing function can be used to get a distinct output for two distinct inputs. The output of Cantor pairing function is a unique natural number 'n' which is a representation of a and b having direct succession relation. It is invertible, we can get back value of a and

¹<http://in.mathworks.com/help/matlab/ref/arrayfun.html>

b from 'n'. The output 'DirectSuccession' in Figure 3.4(c) is of the same size as input arrays and each cell in it stores natural number which is the representation of two activities at the corresponding cells of input arrays having direct succession relation.

$$[B1,...,Bm] = \text{arrayfun}(\text{func}, A1,..., An) \quad (3.3)$$

$$n = \langle a, b \rangle = 1/2(a+b)(a+b+1) + b \quad (3.4)$$

3.4 Graphics Processing Unit (GPU)

While a CPU has a handful number of cores, GPU has a large number of cores along with dedicated high speed memory [32]. Differences between CPU and GPU architecture can be analysed from Figure 3.8 [2]. CPU has larger cache with less number of CUs (Control Unit) and ALUs (Arithmetic Logic Unit) and is designed for serial processing [2]. Whereas GPU has more number of ALUs and CUs that helps in parallel computing of large computation intensive problem [2]. For data to be computed on GPU it has to be sent to GPU and is brought back if the results are required to be accessed by CPU.

The requirements of a program to execute and make use of GPU for better speed performance are that it should be computationally intensive and massively parallel [33]. GPUs perform poor when given a piece of code that involves logical branching. They are meant for doing simple scalar (addition, subtraction, multiplication, division) arithmetic tasks by hundreds of threads running in parallel [33]. While working with GPU, one bottleneck can be transferring the data to and fro from memory as there is a PCI Express (Peripheral Component Interconnect Express) bus through which GPU is connected to CPU, thus memory access is not fast when compared with CPU [34]. Some portions of Alpha Miner algorithm fits the basic criteria to process them on GPU. We identify independent loops in algorithm as discussed in Section 3.2, out of which direct succession relation consumes major part of the running time of the algorithm and can be offloaded to GPU. This part does not involve much of branching across its code and its

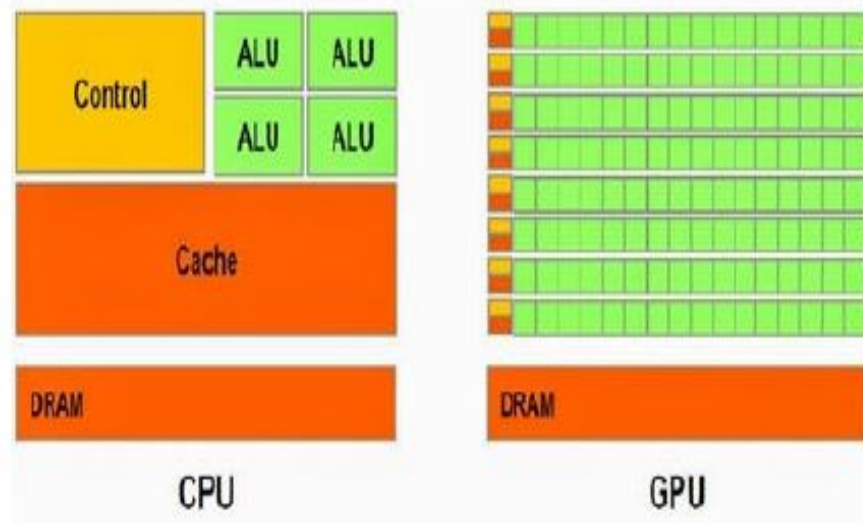


Figure 3.8: Computational Resources of CPU and GPU [2].

computation time far exceeds the data transfer time to and fro from GPU . For MATLAB program's GPU can be used in ways¹ such as:

- (a) Calling some of the GPU enabled MATLAB functions like `fft`, `filter` etc.
- (b) By performing element wise operations through functions like `arrayfun`, `bsxfun` etc.
- (c) By creating and running the kernal of available CUDA file from MATLAB.

Among these, we apply the element wise operations to determine the direct succession relation. GPU only works with numbers (signed and unsigned integers, single-precision and double-precision floating point) thus activities across input file are converted to distinct integers. The data type that works with GPU is `gpuArray`. In Equation 3.5 `gpuArray` copies numeric array B to A and returns a `gpuArray` object. Similarly in Figure 3.4(d), the 'InputFile' i.e. the sequential event log of Alpha Miner algorithm is transferred on GPU device using `gpuArray`. As shown in Figure 3.4(d), to compute the direct succession, first the 'InputFile' is read and shifted to left by 1 to get 'ShiftedFile'. Shifting the file to the left gives immediate successor for each corresponding activity present in the 'InputFile'. We make use of `arrayfun` which is also available for `gpuArray` arguments to do

¹<http://in.mathworks.com/discovery/matlab-gpu.html>

element wise operations on two large arrays. **arrayfun**¹ executes on GPU instead of CPU when one of its arguments is already on GPU. Here, to execute **arrayfun** on GPU, 'InputFile' is already transferred to GPU before. **arrayfun** on GPU executes in the same manner as it executes on CPU. The call to **arrayfun** on GPU is massively parallelized [35]. Using **arrayfun** one call is made to parallel GPU operation that performs the entire calculation instead of making separate calls for each two elements. Also the memory transfer overheads are incurred just once instead on each individual operation. To uniquely encode the pair of activities a, b having direct succession, $a \succ_L b$ we use Cantor pairing function [29] [30] [31] which is passed through **arrayfun**. We make use of Cantor pairing function instead of other functions like concatenation is because GPU only support scalar operations (addition, subtraction, multiplication, division). Cantor pairing function itself involves addition, multiplication and division as shown in Equation 3.4 and thus is suitable for use on GPU. Thus on GPU multiple threads run in parallel and encode activities a, b with $a \succ_L b$ to a unique natural number using Equation 3.4. The output array 'DirectSuccession' of **arrayfun** is of the same size as input arrays. Each cell of the output array stores the representation for two activities that are in the corresponding cells in input arrays and having direct succession relation. The results are brought back to CPU through **gather**. Equation 3.6 shows the syntax of **gather** in which array A from GPU is moved to CPU and stored as array B on CPU. Similarly as shown in Figure 3.4(d), output 'DirectSuccession' array on GPU is brought to CPU using **gather**.

$$A = \text{gpuArray}(B) \quad (3.5)$$

$$B = \text{gather}(A) \quad (3.6)$$

¹<http://in.mathworks.com/help/distcomp/arrayfun.html>

4

Experimental Dataset

For observing and comparing the performance of different types of parallelism, we use 2 real world datasets- Business Process Intelligence 2013 (BPI 2013)¹ and Business Process Intelligence 2014 (BPI 2014)². BPI 2013 Dataset consists of logs from Volvo IT Belgium of its incident and problem management system called VINST. We use VINST case incidents log as shown in Figure 4.1. As displayed in the Figure 4.1, we select 'SRNo' column which represents the Case Id, 'Timestamp' column which represent beginning time of each activity and 'Substatus' column that represents the activity executed in the process. We create a 'SubstatusNo' column which encodes activities to distinct integers. For all the datasets, we generate and use the sequential representation of event log in which activities belonging to same Case Id are arranged in accordance to their time occurrence. Figure 4.2 shows the statistical analysis for BPI 2013 dataset, displaying the case/trace duration vs count of cases. The log has 13 unique activities, 7554 traces, 65533 events. The median case duration is 10.5 days and mean case duration is 12.4 days.

¹<http://www.win.tue.nl/bpi/2013/challenge>

²<http://www.win.tue.nl/bpi/2014/challenge>

SRNo	Timestamp	Status	Substatus	Stlevel	Area	Supsteam	Impact	Product	Code	Country	Owner	SubstatusNo
1-364285768	2010-04-20T10:07:19+01:00	Accepted	Assigned	A2_5	Org line A2	V5 3rd	Medium	PROD582	fr	France	Anne Claire	4
1-364285768	2012-04-11T16:11:17+01:00	Accepted	In Progress	A2_5	Org line A2	V5 3rd	Medium	PROD582	fr	France	Sarah	1
1-364285768	2012-04-11T16:11:25+01:00	Accepted	Assigned	A2_5	Org line A2	V5 3rd	Medium	PROD582	fr	France	Sarah	4
1-364285768	2012-05-03T10:10:10+01:00	Accepted	In Progress	A2_5	Org line A2	V5 3rd	Medium	PROD582	fr	France	Loic	1
1-364285768	2012-05-03T10:10:12+01:00	Completed	Resolved	A2_5	Org line A2	V5 3rd	Medium	PROD582	fr	France	Loic	3
1-364285768	2012-05-11T00:26:15+01:00	Completed	Closed	A2_5	Org line A2	V5 3rd	Medium	PROD582	fr	France	Siebel	5
1-467153946	2011-01-31T11:12:22+01:00	Accepted	In Progress	V3_2	Org line C	S42	Medium	PROD453	se	Sweden	Adam	1
1-467153946	2011-01-31T11:18:44+01:00	Accepted	In Progress	V3_2	Org line C	S42	Medium	PROD453	se	Sweden	Adam	1
1-467153946	2011-01-31T11:19:05+01:00	Queued	Awaiting Assignment	V3_2	Org line C	N52 2nd	Medium	PROD453	se	Sweden	Adam	2
1-467153946	2011-01-31T14:37:55+01:00	Accepted	Wait - User	V3_2	Org line C	N52 2nd	Medium	PROD453	se	Sweden	Denny	6
1-467153946	2011-02-03T08:28:58+01:00	Queued	Awaiting Assignment	C_6	Org line C	O3 3rd	Medium	PROD453	se	Sweden	Denny	2

Figure 4.1: Dataset BPI 2013.



Figure 4.2: Case Duration for BPI 2013.

BPI 2014 contains Rabobank Group ICT data. The data contain events of Information Technology Infrastructure Library that aligns IT services with business needs. Detail Incident Activity log is used. As shown in the Figure 4.3, we select 'IncidentID' column which represents the Case Id, 'Datestamp' column which represents the starting time of each activity and 'IncidentActivity_Type' column that represents the activity executed in the process. We create a 'Inciden-

tActivity_Type_Number' column which encodes activities to distinct integers. It consists of 39 unique activities, 46616 traces and 466737 events. Shown in Figure 4.4 is the statistical analysis of BPI 2013 dataset. We conduct experiments on publicly available dataset so that our experiments can be replicated and used for benchmarking or comparisons.

IncidentID	DateStamp	IncidentActivity_Number	IncidentActivity_Type	AssignGroup	KMnumber	InteractionID	IncidentActivity_Type_Number
IM0005084	14-10-2013 11:16:11	001A5681229	External update	TEAM0105	KM0000480	SD0012621	36
IM0005084	14-10-2013 11:17:42	001A5681230	External update	TEAM0105	KM0000480	SD0012621	36
IM0005084	14-10-2013 11:17:47	001A5681231	Closed	TEAM0105	KM0000480	SD0012621	5
IM0005084	14-10-2013 11:16:11	001A5681227	Assignment	TEAM0105	KM0000480	SD0012621	4
IM0005085	14-10-2013 12:25:07	001A5683858	Operator Update	TEAM0008	KM0000223	SD0012613	3
IM0005085	14-10-2013 14:39:47	001A5686340	Caused By CI	TEAM0044	KM0000223	SD0012613	6
IM0005085	14-10-2013 14:39:47	001A5686339	Closed	TEAM0044	KM0000223	SD0012613	5
IM0005085	14-10-2013 11:22:42	001A5682528	Operator Update	TEAM0008	KM0000223	SD0012613	3
IM0005085	14-10-2013 11:21:33	001A5682527	Open	TEAM0008	KM0000223	SD0012613	16
IM0005086	14-10-2013 14:25:29	001A5683430	Update	TEAM0001	KM0001403	SD0012622	7
IM0005086	14-10-2013 14:22:08	001A5683413	Operator Update	TEAM0001	KM0001403	SD0012622	3

Figure 4.3: Dataset BPI 2014.

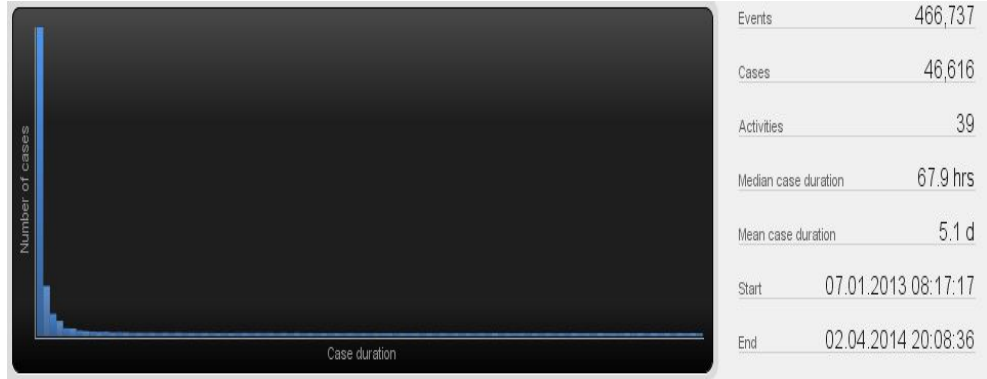


Figure 4.4: Case Duration for BPI 2014.

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). The algorithm of generation of the synthetic data is shown in

Algorithm 1. To imitate the real world event logs, the algorithm first defines direct succession relations between randomly chosen activities leading to formation of causal, parallel or unrelated between all the activities. Then using a half normal distribution with a given mean and standard deviation it randomly generates the length of each trace. A random activity is chosen to be placed at the beginning of the trace. Next activity in the trace is chosen randomly from the set of the direct successors of the current activity.

Algorithm 1: Synthetic Data Generation

Data: Standard deviation s , mean m , trace count c , activity count a

Result: Synthetic data in form of sequential event log

```

1 l=1
2 for i=1 : a do
3     x=l+rand(1,1)*(a-l)
4     x=round(x)
5     for j=1: x do
6         r=l+rand(1,1)*(a-l)
7         r=round(r)
8         Successor(i,j)=r
9 for i=1 : c do
10    x=round(randn(1,1) * s + m)
11    if x < 0 then
12        x=abs(x)
13    r = l+(a-l).*rand(1,1)
14    r=round(r)
15    for j=2: x do
16        n= Successor(r,find(Successor(r,:)))
17        n=datasample(n,1)
18        arr=[arr,n]
19        r=n
20    append arr to output file.
```

Using Algorithm 1, 2 datasets are prepared, dataset A with standard deviation 10, mean 20, activity count 20, shown in Figure 4.5 while dataset B with standard

deviation of 25, mean 50 and activity count 50 as shown in Figure 4.6.

Caseld	Activity_List
1	18,15,16,17,6,12,3,11,11,8,14,9,18,17,3
2	4,14,19,19,2,15,17
3	5,11,4,7,8,16,13,18,17,9,7,17,3,14,18,3,11,4,13,3,4,14
4	5,3,11,1,5,14,10,4,7,8,9,7,9
5	7,8,16,18,18,19,9,18,18,17,9,17,13,3,11,14,20,4,11,17
6	11,14,19,19,9,17,16,20,5,17,16,5,17,11,20,4,7,17,20,6,10,12,17,9,17,20,17,11,20,7,17,20,4,13
7	12,3,14,11,2,8,9,17,10,12,15,16,7
8	7,10,19,15,12,9,17
9	5,14,16,20,5,8,12,16,16,8,9,7,8,6,6,8,9,17,9,7,8,9,18,3,14,8,12,12,15,16,16,3
10	13,18,8,9,7,8,9,18,8
11	9,17,3,18,2,8,14,8,6,6,10,12,9
12	19,9,18,10,4,2,15,16,18,15,16,18,18,3,11,18,17
13	14,8,14,3,18,18
14	7,19,18,3,7,10,19,18,19,2,16,18,2,20,4,11,20,7,10,4,13,7,10
15	3,12

Figure 4.5: Synthetic Dataset A.

Caseld	Activity_List
1	21,9,16,28,48,12,12,1,23,49,4,34,29,33,32,11
2	3,11,29,8,36,15,27,27,46,31,4,18,27,27,14,3,32,13,20,16,42,9,40,16,43,21,32,20,16,12,1,9,25,44,36,16,50,7,14,43,35,44,43,21,17,17,11,41,29,1,30,3,43,21,38,33,20,12,14,28,25,17,17,15,27,50,23,33,36,49,18,10,6,33,
3	10,42,9,44,9,43,21,46,41,12,12,24,38,14,27,7,6,21,24,38,16,12,12,31,4,24,38,32,12,7,41,10,23,49,24,38,23,35,2,26,8,16,48,24,38,2
4	38,21,6,48,27,14,15,2,41,6,42,6,31,19,46,3,6,10,7,32,13,15,35,14,35,14,8,41,10,42,9,27,23,7,6,10,39,15,2,34,43,35,21,14,16,9,33,29,37,42,9,48,25,30,41,10,50,42,9,7,10,13,15,8,16,43,21,23,49,42,6,42,6,47,48,22,48,
5	3,6,36,49,7,17,13,22,43,35,17,33,15,36,11,50,36,19,10,12,12,40,49,24,38,23,30,3,6,34,33,41,22,43,21,42,9,7,11,31,29,6,49,44,12,17,14,3
6	26,39,21,9,40,23,11,40,26,23,7,39,15,26,12,46,46,46,41,24,38
7	13,37,13,26,44,32,28,3,6,30,6,31,43,6,42,9,39,33,15,27,23,11,29,21,23,11,25,25,26,2,34,33,12,17,15,35,48,4,18,22,37,42,9,43,35,15,39,32,11,41,10,26,13,21,42,6,36,15,38,21,24,38,5,49,30,9,18,22,12,44,43,35,48,12,
8	23,33,35,41,30,15,27,39,30,21,42,9,27,14
9	43,6,40,50,36,11,35,4,24,38,47,50,50,25,32,15,27,37,42,6,47,2,45,32,28,25,17,31,29,21,6,21,26,22,36,16,45,31,31,22,16,36,19,15,39,30,3,32,28,3,32,26,35,17,10,16,8,41,6,40,12,12,21,38,15
10	18,10,47,8,32
11	50,30,23,30,2,34,33,25,26,18,22,35,7,40,12,12
12	33,2,1,15,27,50,42,6,21,38,33,34,43,6,49,6,3,43,35,39,46,3,6
13	5,31,25,33,20,16,45,19,45,43,21,23,11,42,9,27,27,8,36,15,3,6,36,49,45,37,15,11,12,1,20,37,42,9,48
14	6,5,14,43,6,33,20,47,2,4,14,32,26,37,13,8,39,15,39,21,1,12,40,44,24,38,39,19,28,28,43,6,40,2,45,19,48,29,48,44,13,41,45,37,22,1,41,32,3,11,16,12,22,24,38,33,47,16,23
15	27,7,41,11,49,8,1,6,10,48,34,29,48,22,15,4,14,27,27,37,45,42,9,18,10,42,9,44,12,12,15,38,39,21,44,32,15,11,48,22,28,19,9,9,27,37,23,49,9,35,18,10,50,23,33,6,49,7,14,25,1

Figure 4.6: Synthetic Dataset B.

To get insights into performances of different implementations with variation of datasize, each dataset is recorded for 5 increasing trace count values. For CPU, datasets A and dataset B are generated with trace counts 500, 2000, 8000, 32000 and 128000. Since GPUs are designed to work with large data [33], we generate bigger datasets for it. For GPU, datasets A and dataset B are generated with trace counts 10000, 50000, 250000, 1250000 and 6250000. Table 4.1 describes the details of experimental datasets.

Table 4.1: Details of Experimental Datasets.

Datasets	No. of Events	Trace Count	Activity Count
BPI 2013	65533	7554	13
BPI 2014	466737	46616	39
Dataset A	9991	500	20
	38374	2000	20
	155296	8000	20
	192464	10000	20
	614781	32000	20
	956213	50000	20
	2452823	128000	20
	4788140	250000	20
	23969927	1250000	20
	119841817	6250000	20
Dataset B	24777	500	50
	97589	2000	50
	395925	8000	50
	495522	10000	50
	1587454	32000	50
	2472278	50000	50
	6351629	128000	50
	12375747	250000	50
	61848409	1250000	50
	308921230	6250000	50

5

Experimental Settings and Results

Table 5.1: 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	96 GB
Operating System	Linux, 64 bit
Graphics Card	NVIDIA Tesla K40c
GPU Cores	2880
GPU Memory	12 GB
MATLAB Version	R2014b

The parameters of computer and GPU used for testing are shown in Table 5.1. To get the accurate execution timings of programs, we perform the experiments in isolation. We measure timings using two MATLAB functions, namely `tic` which starts stopwatch timer and `toc` that displays the elapsed time. We calculate the speedup (S) using the Equation 5.1 where T_{old} is old execution time and T_{new} is the new execution time with improvement [36]. We set the speedup value to $1\times$ for implementations whose execution time is considered as T_{old} . For all the

implementations, we record time that includes both the computations involved and data transfers to and fro from workers or GPU. Implicit multithreading in MATLAB uses threads equal to number of logical processor with hyperthreading¹ 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. Each worker is a thread on a CPU core. To see the performance results with varying number of workers (cores), we carry the `parfor` implementation on 2, 4, 6, 8 and 10 physical cores. We run the program using `parfor` after it is connected to specific number of workers, not recording the time to start the parallel pool.

$$S = T_{old}/T_{new} \quad (5.1)$$

Table 5.2: Execution Time (sec) of Single-threaded, `parfor` and Multi-threaded implementations of Alpha Miner Algorithm on CPU.

Datasets	Trace Count	Activity Count	Single-threaded CPU	parfor CPU	Multi-threaded CPU
BPI 2013	7554	13	27.93	17.93	8.26
BPI 2014	46616	39	423.15	149.4	69.6
Dataset A	500	20	1.87	2.88	1.05
	2000	20	4.08	3.92	1.85
	8000	20	14.53	10.51	4.93
	32000	20	75.25	36.71	18.49
	128000	20	825.5	154.6	74.32
Dataset B	500	50	9.48	9.03	7.21
	2000	50	14.08	11.78	9.313
	8000	50	41.7	29.38	13.38
	32000	50	195.31	87.55	46.09
	128000	50	2247.5	371.9	212.87

¹<http://www.intel.in/content/www/in/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html>

Table 5.2 reports the end to end timing observed on CPU for 3 implementations of the Alpha Miner algorithm, namely Single-threaded, **parfor** (2 workers) and Multi-threaded.

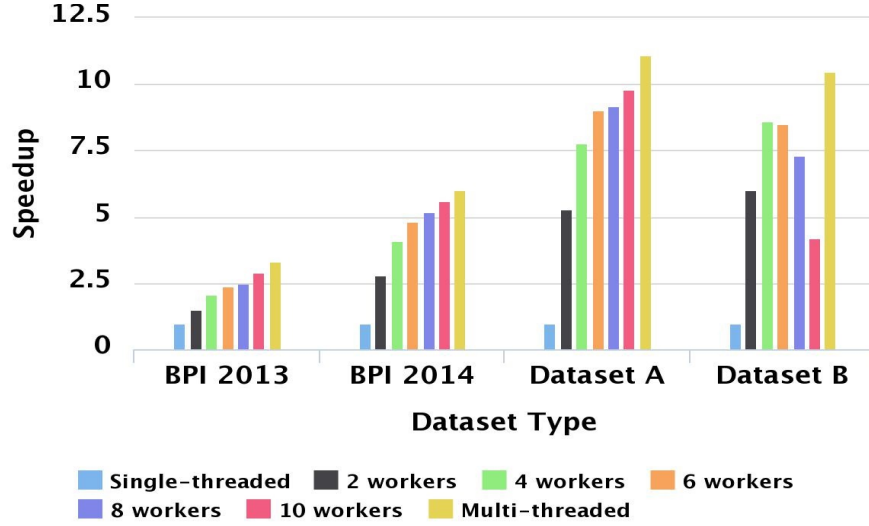


Figure 5.1: Speedup gain by **parfor** and Multi-threaded Parallelism on CPU across various datasets.

Figure 5.1 and Figure 5.2 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 Figure 5.1 speedup is shown at highest trace count 128000 for dataset A and B. As shown in Figure 5.1, using 2 workers we obtain good speedup values which increase with increase in datasize, ranging from $1.55\times$ in the smallest dataset (BPI 2013) to $6.04\times$ in the largest dataset (dataset B). We expect the performance to double using 4 workers but it ranges from minimum value of $2.19\times$ (BPI 2013) to maximum of $8.60\times$ (dataset B). Similar effect is observed with further increase in workers with speedup values increasing marginally. Marginal increase happens in performance as adding more workers leads to more communication overheads eventually reducing the gains of parallelism¹. In fact, over the largest dataset B, performance degrades with increase in number of workers. Although due to larger size dataset B involves

¹<http://in.mathworks.com/company/newsletters/articles/improving-optimization-performance-with-parallel-computing.html>

more computations, overheads of calling workers, distributing work 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.

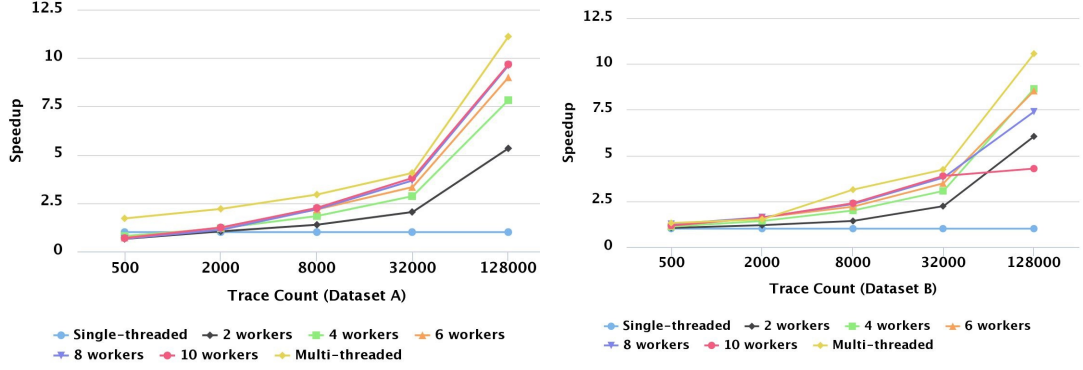


Figure 5.2: Speedup gain by `parfor` and Multi-threaded Parallelism on CPU with varying dataset size.

Communication overheads also outweighs the benefits of parallelism when computations are too less i.e. while working with smaller dataset. Figure 5.2 reveals that speedup value comes to be less than $1 \times$ of 500 trace count in dataset A for 10 workers. In Figure 5.2, speedup values continue to increase with increase in both trace count and worker within dataset A and dataset B till the time computation time outweighs communication overheads with workers. As observed from Figure 5.1 and Figure 5.2, highest speedup is always achieved through multi-threaded parallelism. This can be attributed to the fact that multithreading 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. While `parfor` offers data level parallelism, multithreading does instruction level parallelism. We observe that in MATLAB multithreading parallelism is triggered only by performing element wise operation on large matrices, whereas inside `parfor` loop we can write any type of code. We ensure not to send too much data inside the `parfor` loop to the workers to keep data transfer costs minimum. Multithreading parallelism is initiated by just a single line of code like done through `arrayfun` thus no problem of data transfers is involved. We see that in multithreading

parallelism with larger data resulting in more computations, better performance is achieved, the same trend is observed in **parfor** but not with every count of workers. Multithreading parallelism itself creates threads in the program, unlike in **parfor** where a user has to set up a parallel pool and specify the worker count. **parfor** does utilise number of cores of the CPU as specified by the user. As shown in Figure 5.3, with 20 logical cores available on machine, CPU utilisation grows approximately by 10 % with increase in 2 workers. Use of multithreading, does not guarantee that all cores will be optimally used. It is upto operating system (OS) as how to partition the work load to different cores. OS may make efficient use of the idle time of a CPU core to run multiple threads in same core, instead of distributing to different cores¹.

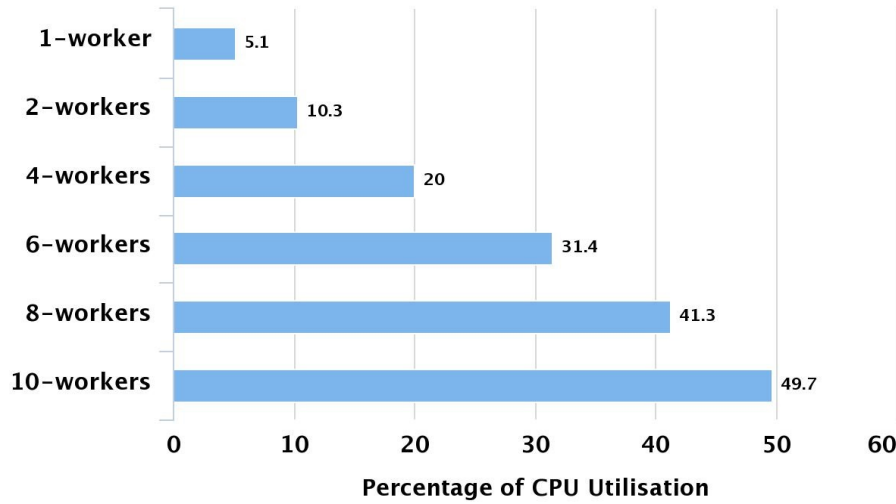


Figure 5.3: CPU Utilisation with varying workers.

As mentioned in Section 3.2, that **parfor** is applied on 3 different independent parts of the program. From Figure 5.4 (dataset A and dataset B taken at 32000 trace count) we observe that both the footprint building and maximal set pairs generation do not contribute significantly in single-threaded program's execution time and thus in overall speedup. Direct succession holds major percentage in the program's running time in every dataset and thus reduction in its execution

¹http://cache-www.intel.com/cd/00/00/01/77/17705_htt_u_ser_guide.pdf

time using `parfor` leads to gain in speedup. Thus, countering the bottleneck only helps in attaining a good speedup.

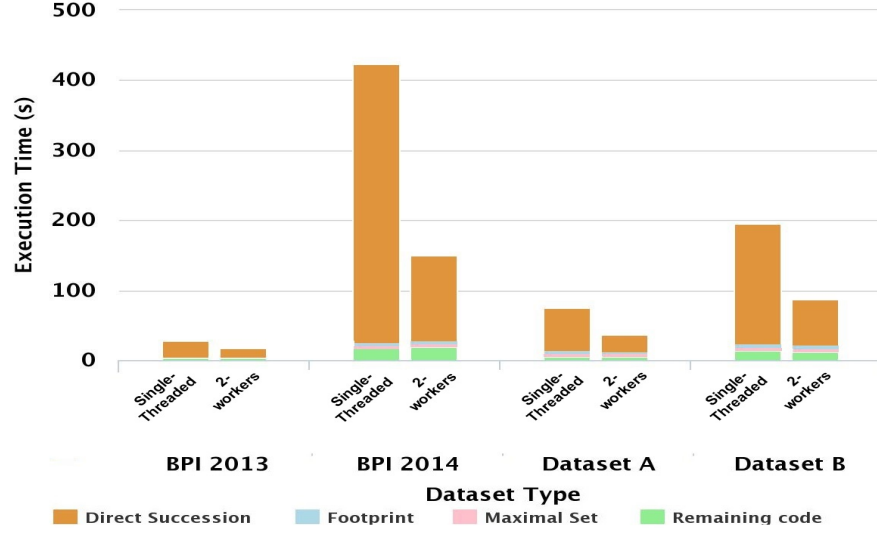


Figure 5.4: Performance of different `parfor` loops across datasets.

Table 5.3: Execution Time (sec) of Multi-threaded CPU and GPU implementations of Alpha Miner Algorithm.

Datasets	Trace Count	Activity Count	Multi-threaded CPU	GPU
BPI 2013	7554	13	8.26	0.96
BPI 2014	46616	39	69.6	4.64
Dataset A	10000	20	6	1.32
	50000	20	24.23	1.87
	250000	20	134.6	4.39
	1250000	20	672.9	18.58
	6250000	20	3791	96.42
Dataset B	10000	50	18.11	7.55
	50000	50	76.94	11.87
	250000	50	330.15	16.00
	1250000	50	1722.6	53.68
	6250000	50	9137.6	266.14

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. The end to end execution time recorded for multi-threaded CPU and GPU implementation is given in Table 5.3. Displayed in Figure 5.5 is the multi-threaded Alpha Miner CPU code on GPU leading to a speedup as high as $39.3\times$. As can be seen from Figure 5.5 significant speedups are achieved across datasets. In Figure 5.5 for datasets A and B, speedup is calculated at highest data point 6250000 to observe the performance at the peak.

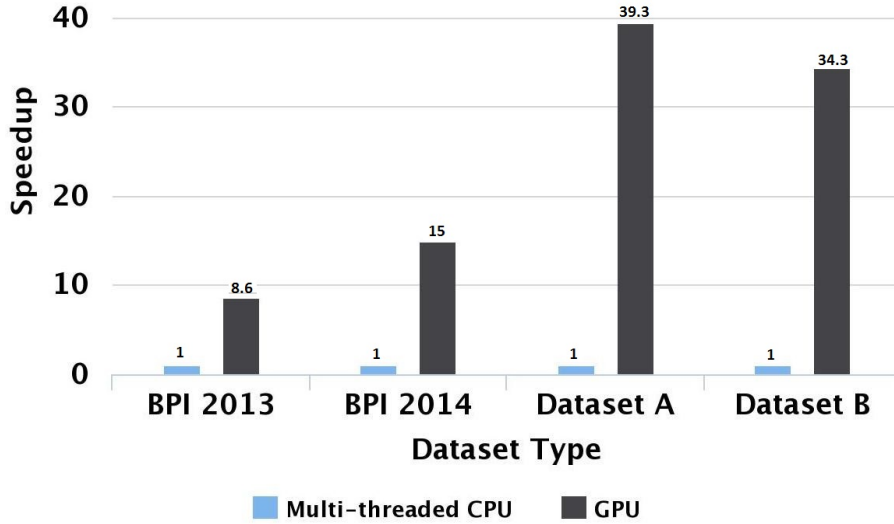


Figure 5.5: Speedup gain by GPU across various datasets.

Figure 5.6 reveals that within a given dataset with increase in trace count, GPU performance improves. We see that with increase in datasize, the penalty of overheads of data transfers to and from GPU becomes smaller in context of larger computations hence speedup value improves with increase in trace count. Therefore, we obtain highest speedup at the largest trace count value for both the datasets. Hence, we believe that the performance will further increase with increase in trace count. At trace count 6250000 in dataset B, the size of array transferred on GPU exceeds the memory limit of GPU. Thus the array is broken into two parts and computed separately to get the results. Hence we infer that GPU memory limits should be taken into account while working with GPU. The

line chart in Figure 5.6 reveals that performance on dataset B grows relatively slower than on dataset A. Alpha Miner on dataset B 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 fro 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.

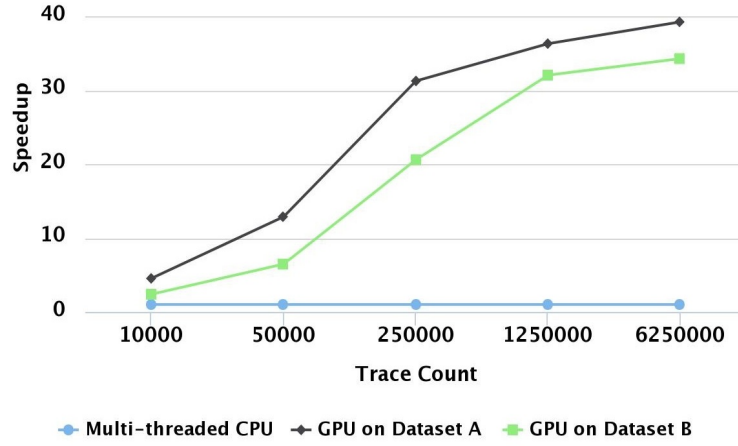


Figure 5.6: Speedup gain by GPU with varying dataset size.

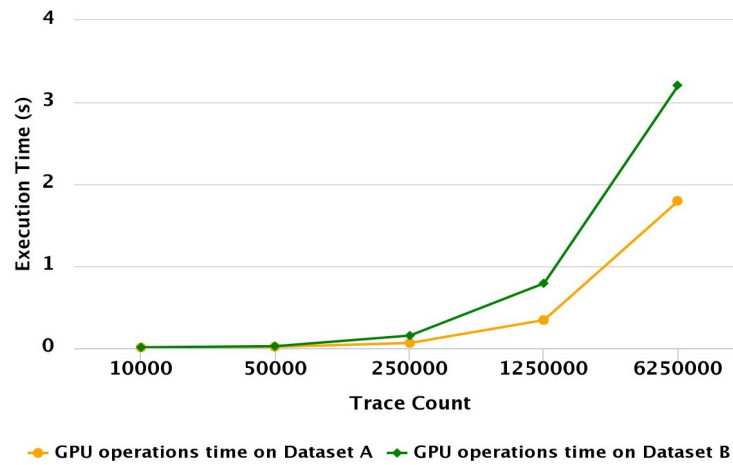


Figure 5.7: Time taken to perform all GPU related tasks (data transfer, gather and arrayfun) on Dataset A and Dataset B.

The time required for performing GPU related tasks like transfer and gather of data to and fro from GPU and doing computations on it is almost same for lower trace count points for both the datasets as shown in Figure 5.7. At higher dataset size (1250000) only we see some differences between the time taken by the two. The total time taken to perform GPU related tasks as shown in Figure 5.7 is in very small percentage ranging to maximum $\approx 2\%$ in dataset A and $\approx 1\%$ in dataset B, with respect to end to end time reported in Table 5.3 to run the entire program using GPU. Thus, with GPU operations not taking too significant time, we are encouraged to test even bigger datasets on GPU. With small amount of timings required to do GPU intensive operations by bigger datasets and in return getting parallelism over hundreds of cores, we can expect higher speedup gains on GPU for Alpha Miner algorithm for any other dataset also.

6

Limitations and Future Work

Parallelization of Alpha Miner has been done on MATLAB which is a high level language and has an expensive license. Using proposed parallel approach for Alpha Miner algorithm, work can be implemented on low level languages also. Apart from taking into account certain limitations and rules of different parallelism techniques, achieving good speedup through parallelization in MATLAB is a simple task. But we see that MATLAB provides just a black box for doing parallelization. It internally calls the subroutines written in C, C++ etc. hiding from programmer all details about data parallelism exploration. Writing directly into C, C++ can let user to have a better control over the parallelization and obtain better speedup. Also for multithreading and GPU parallelism MATLAB is suitable for providing speedups only for numerical problems working on large matrices. Partitioning of data for performing parallelization on Alpha Miner can be done by other ways also and parallelized code can be tested for more optimizations example multi-threaded function `bsxfun` can be used in place of `arrayfun` and performances can be compared to find out the better function. We can test our models on more diverse real life datasets. Performance of parallelism models can also be tested and compared on different machines. Alpha Miner can be tested for better performance by the use of more powerful GPU cards. Due to expensive license problem of MATLAB Distributed Computing Server, we are not able to extend the research to a grid of computers.

7

Conclusion

We conduct a series of experiments on synthetic and real-world datasets to observe the performance of Alpha Miner algorithm on parallelization. We make use of MATLAB Parallel Computing Toolbox for constructs such as `parfor` to distribute computations across multi-cores of CPU and also for accessing GPU. We find multi-threaded functions to be more efficient than the explicit parallelism done through `parfor` construct. The speedup achieved using implicit multithreading is always higher than using `parfor`. Thus, use of shared memory in case of implicit parallelism proves to be more beneficial than creating separate processes in `parfor`. The performance of `parfor` keeps improving with increase in number of workers till the time communication overheads associated with the workers are not significant. We achieve the highest performance while doing parallelization of Alpha Miner over GPU with speedup as high as $39.3\times$ is obtained. Within a given dataset, the parallelization performance over GPU improves with increase in datasize. Thus, we accelerate Alpha Miner algorithm using different parallelism techniques and achieve the maximum speedup by utilising the potential of GPU computing. Due to our proposed parallelization of computing the direct succession relation, Alpha Miner algorithm can now be worked on larger size event logs. The proposed approach of parallelization of building footprint matrix and finding maximal set pairs can enable to work on higher count of activities.

References

- [1] WIL VAN DER AALST. **Process Mining: Overview and Opportunities.** *ACM Trans. Manage. Inf. Syst.*, **3**(2):7:1–7:17, July 2012. vi, 1, 2, 3
- [2] V.H. NAIK AND C.S. KUSUR. **Analysis of performance enhancement on graphic processor based heterogeneous architecture: A CUDA and MATLAB experiment.** In *Parallel Computing Technologies (PAR-COMPTech)*, 2015 National Conference on, pages 1–5, Feb 2015. vi, 24, 25
- [3] WIL M. AALST. **Transactions on Petri Nets and Other Models of Concurrency II.** chapter Process-Aware Information Systems: Lessons to Be Learned from Process Mining, pages 1–26. Springer-Verlag, Berlin, Heidelberg, 2009. 1
- [4] JAMES L. PETERSON. **Petri Nets.** *ACM Comput. Surv.*, **9**(3):223–252, September 1977. 1
- [5] WIL VAN DER AALST, ARYA ADRIANSYAH, AND BOUDEWIJN VAN DONGEN. **Causal Nets: A Modeling Language Tailored Towards Process Discovery.** In *Proceedings of the 22Nd International Conference on Concurrency Theory*, CONCUR’11, pages 28–42, Berlin, Heidelberg, 2011. Springer-Verlag. 1
- [6] W. M. P. VAN DER AALST, H. A. REIJERS, A. J. M. M. WEIJTERS, B. F. VAN DONGEN, A. K. ALVES DE MEDEIROS, M. SONG, AND H. M. W. VERBEEK. **Business Process Mining: An Industrial Application.** *Inf. Syst.*, **32**(5):713–732, July 2007. 2

-
- [7] WIL VAN DER AALST, TON WEIJTERS, AND LAURA MARUSTER. **Workflow mining: Discovering process models from event logs.** *Knowledge and Data Engineering, IEEE Transactions on*, **16**(9):1128–1142, 2004. 4, 11
 - [8] GABRIELE JOST, HAO-QIANG JIN, DIETER ANMEY, AND FERHAT F HATAY. **Comparing the openmp, mpi, and hybrid programming paradigm on an smp cluster.** 2003. 8
 - [9] JOHN D OWENS, MIKE HOUSTON, DAVID LUEBKE, SIMON GREEN, JOHN E STONE, AND JAMES C PHILLIPS. **GPU Computing.** *Proceedings of the IEEE*, **96**(5):879–899, 2008. 8
 - [10] JOHN NICKOLLS, IAN BUCK, MICHAEL GARLAND, AND KEVIN SKADRON. **Scalable parallel programming with CUDA.** *Queue*, **6**(2):40–53, 2008. 8
 - [11] KUO-YI CHEN, J MORRIS CHANG, AND TING-WEI HOU. **Multithreading in Java: Performance and scalability on multicore systems.** *Computers, IEEE Transactions on*, **60**(11):1521–1534, 2011. 8
 - [12] WIL M. P. VAN DER AALST AND CHRISTIAN W. GÜNTHER. **Finding Structure in Unstructured Processes: The Case for Process Mining.** In *Seventh International Conference on Application of Concurrency to System Design (ACSD 2007), 10-13 July 2007, Bratislava, Slovak Republic*, pages 3–12, 2007. 11
 - [13] AJMM WEIJTERS, WIL MP VAN DER AALST, AND AK ALVES DE MEDEIROS. **Process mining with the heuristics miner-algorithm.** *Technische Universiteit Eindhoven, Tech. Rep. WP*, **166**:1–34, 2006. 11
 - [14] BOUDEWIJN F VAN DONGEN AND WIL MP VAN DER AALST. **Multi-phase process mining: Building instance graphs.** In *Conceptual Modeling–ER 2004*, pages 362–376. Springer, 2004. 11, 12
 - [15] BOUDEWIJN F VAN DONGEN AND WIL MP VAN DER AALST. **Multi-phase process mining: Aggregating instance graphs into EPCs and Petri nets.** In *PNCWB 2005 workshop*, pages 35–58. Citeseer, 2005. 11, 12

-
- [16] ARMIN AHMADZADEH, REZA MIRZAEI, HATEF MADANI, MOHAMMAD SHOBEIRI, MAHSA SADEGHI, MOHSEN GAVAH, KIANOUSH JAFARI, MOHSEN MAHMOUDI AZNAVEH, AND SAEID GORGIN. **Cost-efficient implementation of k-NN algorithm on multi-core processors.** In *Formal Methods and Models for Codesign (MEMOCODE), 2014 Twelfth ACM/IEEE International Conference on*, pages 205–208. IEEE, 2014. 12
- [17] TAKAZUMI MATSUMOTO, EDWARD HUNG, AND MANLUNG YIU. **Parallel outlier detection on uncertain data for GPUs.** *Distributed and Parallel Databases*, pages 1–31, 2014. 12
- [18] LI NAN, GAO PENG DONG, LU YONGQUAN, AND YU WENHUA. **The Implementation and Comparison of Two Kinds of Parallel Genetic Algorithm Using Matlab.** In *Distributed Computing and Applications to Business Engineering and Science (DCABES), 2010 Ninth International Symposium on*, pages 13–17, Aug 2010. 12
- [19] JOHN A. STRATTON, SAMS. STONE, AND WEN-MEI W. HWU. **MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs.** In JOSNELSON AMARAL, editor, *Languages and Compilers for Parallel Computing*, 5335 of *Lecture Notes in Computer Science*, pages 16–30. Springer Berlin Heidelberg, 2008. 12
- [20] SUNGPACK HONG, T. OGUNTEBI, AND K. OLUKOTUN. **Efficient Parallel Graph Exploration on Multi-Core CPU and GPU.** In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 78–88, Oct 2011. 12
- [21] LUKASZ LIGOWSKI AND WITOLD RUDNICKI. **An efficient implementation of Smith Waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databases.** In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8. IEEE, 2009. 12

- [22] KHEDIJA AROUR AND AMANI BELKAHLA. **Frequent Pattern-growth Algorithm on Multi-core CPU and GPU Processors.** *CIT*, **22**(3):159–169, 2014. 12
- [23] J.A. STUART AND J.D. OWENS. **Multi-GPU MapReduce on GPU Clusters.** In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 1068–1079, May 2011. 12
- [24] MIAN LU, YUWEI TAN, GE BAI, AND QIONG LUO. **High-performance Short Sequence Alignment with GPU Acceleration.** *Distrib. Parallel Databases*, **30**(5-6):385–399, October 2012. 13
- [25] LCM PAULA AND ANDERSON DA SILVA SOARES. **Parallel implementation of the BiCGStab (2) method in GPU using cuda and Matlab for solution of linear systems.** *Journal of Communication and Computer*, **11**:339–346, 2014. 13
- [26] VIPIN KUMAR. *Introduction to Parallel Computing.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002. 18
- [27] G. S. ALMASI AND A. GOTTLIEB. *Highly Parallel Computing.* Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1989. 18
- [28] DESMOND J. HIGHAM AND NICHOLAS J. HIGHAM. *Matlab Guide.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2005. 23
- [29] GEORG CANTOR. *Contributions to the Founding of the Theory of Transfinite Numbers.* Dover, New York, 1955. Original year was 1915. 23, 26
- [30] G. CANTOR. **Ein Beitrag zur Mannigfaltigkeitslehre.** *Journal fr die reine und angewandte Mathematik*, **84**:242–258, 1877. 23, 26
- [31] MERI LISI. **Some remarks on the Cantor pairing function.** *Le Matematiche*, **62**(1), 2007. 23, 26
- [32] WEN-MEI W. HWU. *GPU Computing Gems Emerald Edition.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011. 24

REFERENCES

- [33] JUNG W. SUH AND YOUNGMIN KIM. *Accelerating MATLAB with GPU Computing: A Primer with Examples*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013. 24, 32
- [34] RAVI BUDRUK, DON ANDERSON, AND ED SOLARI. *PCI Express System Architecture*. Pearson Education, 2003. 24
- [35] J.W. SUH AND Y. KIM. *Accelerating MATLAB with GPU Computing: A Primer with Examples*. Morgan Kaufmann. Elsevier/Morgan Kaufmann, 2013. 26
- [36] JOHN L. HENNESSY AND DAVID A. PATTERSON. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3 edition, 2003. 33
- [37] NATHAN L PARRISH. *Accelerating Lamberts problem on the gpu in Matlab*. PhD thesis, California Polytechnic State University, San Luis Obispo, 2012.
- [38] A. J. M. M. WEIJTERS AND A. K. ALVES DE MEDEIROS. **Process Mining with the HeuristicsMiner Algorithm**.
- [39] WIL VAN DER AALST. **Process Mining: Making Knowledge Discovery Process Centric**. *SIGKDD Explor. Newsl.*, **13**(2):45–49, May 2012.