Parallelization of Transition Counting for Process Mining on Multi-core CPUs and GPUs

Diogo R. Ferreira and Rui M. Santos

Instituto Superior Técnico (IST), University of Lisbon, Portugal
{diogo.ferreira,rui.miguel.santos}@tecnico.ulisboa.pt

Abstract. Many process mining tools and techniques produce output models based on the counting of transitions between tasks or users in an event log. Although this counting can be performed in a forward pass through the event log, when analyzing large event logs according to different perspectives it may become impractical or time-consuming to perform multiple such passes. In this work, we show how transition counting can be parallelized by taking advantage of CPU multi-threading and GPU-accelerated computing. We describe the parallelization strategies, together with a set of experiments to illustrate the performance gains that can be expected with such parallelizations.

1 Introduction

Transition counting is the basis of many process mining techniques. For example, the α-algorithm [1] uses $a >_W b$ to denote the transition between two consecutive tasks in a trace, and the HeuristicsMiner [2] uses the count of such transitions $|a >_W b|$ to derive a dependency graph from the event log. Popular process mining tools, such as Disco [3], also display the discovered model as a graph where the arcs between activities are labeled with transition counts. Transition counting is therefore an essential task in several control-flow algorithms.

Also in the organizational perspective, transition counting plays a central role in extracting sociograms based on metrics such as *handover of work* and *working together* [4, 5]. The handover of work metric requires counting the transitions between successive users who participate a case. The working together metric (also known as *joint cases*) counts the number of cases in which a given pair of users have worked together (not necessarily in direct succession). This too can be regarded as a problem of transition counting, where both direct and indirect transitions between users in a case are considered.

Parallel computing [6] offers tremendous possibilities to improve the performance of process mining techniques, especially when event logs become increasingly large. The BPI Challenge 2016[1] is the first in its series (since its inception in 2011) where the event logs to be analyzed exceed 1 GB in size. But as early as 2009, we were already experiencing some difficulties in processing a set of event logs that amounted to 13 GB in total size [7]. Clearly, for such large event logs,

[1] http://www.win.tue.nl/bpi/doku.php?id=2016:challenge

it becomes impractical to explore and analyze them by running multiple passes over the entire event log in a single-threaded fashion.

The availability of multi-core CPUs and the trend towards powerful GPUs (Graphics Processing Units) that can be used for general-purpose computing create the opportunity to leverage those technologies to accelerate the processing of large event logs. There are certainly many techniques that can potentially benefit from such parallelization, but in this work we focus on the essential task of counting transitions. Although the problem might appear to be simple, we will see that its parallelization involves some challenges and trade-offs. In particular, the parallelization on the GPU is fundamentally different from that on a multi-core CPU. We present a viable strategy to perform both.

It should be noted that we are not the first to attempt such parallelization. In the BPI Workshop 2015, there was a work on the parallelization of the α-algorithm [8]. However, such work was based on a single, high-level construct provided by MATLAB (specifically, the parallel for-loop). Here, we go much deeper into the parallelization by controlling the execution of threads at the lowest level of detail. On the CPU, we use POSIX threads [9], and on the GPU we use NVIDIA's CUDA technology and programming model [10].

Section 2 introduces a sample process and event log. Section 3 describes the algorithms that are to be parallelized. Section 4 describes the parallelization in the CPU, and Section 5 describes the parallelization on the GPU; both sections include experiments and results. Finally, Section 6 concludes the paper.

2 An example process

As a running example, and in order to generate variable-size event logs for testing purposes, we use the purchase process from [11]. This process is illustrated in Figure 1. Basically, there are two main branches: if the purchase request is not approved, it is archived and the process ends; if the purchase request is approved, the product is ordered from the supplier, the warehouse receives the product, and the accounting department takes care of payment.

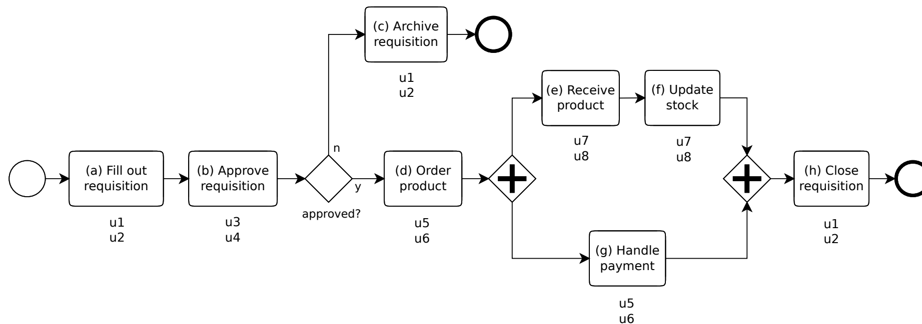

Fig. 1. A simple purchase process

In the lower branch, some activities are performed in parallel, meaning that their execution order is non-deterministic. In addition, each activity is performed by one of two users, depending on who is first available to pick the task at run-time. There are eight users involved in this process, and they are shared by multiple activities, as indicated in Figure 1.

Table 1 shows a sample event log that has been generated from a simulation of this process. In this small example there are only three cases, but for testing purposes we will use many more (up to 10^7 cases). Our analysis will be focusing on the *case id*, *task* and *user* columns.

case id	*task*	*user*	*timestamp*
1	a	u_1	2016-04-09 17:36:47
1	b	u_3	2016-04-11 09:11:13
1	d	u_6	2016-04-12 10:00:12
1	e	u_7	2016-04-12 18:21:32
1	f	u_8	2016-04-13 13:27:41
1	g	u_6	2016-04-18 19:14:14
1	h	u_2	2016-04-19 16:48:16
2	a	u_2	2016-04-14 08:56:09
2	b	u_3	2016-04-14 09:36:02
2	d	u_5	2016-04-15 10:16:40
2	g	u_6	2016-04-19 15:39:15
2	e	u_7	2016-04-20 14:39:45
2	f	u_8	2016-04-22 09:16:16
2	h	u_1	2016-04-26 12:19:46
3	a	u_2	2016-04-25 08:39:24
3	b	u_4	2016-04-29 10:56:14
3	c	u_1	2016-04-30 15:41:22

Table 1. Sample event log

In this example, the events have been sorted by case id and timestamp, as required by the algorithms to be described below. We assume that this sorting has already been done, or can be done as a one-time preprocessing step. When using a common log format such as MXML [12] or XES [13], such sorting step is unnecessary because the events are already grouped by case id.

3 Algorithms

In this work we consider two basic algorithms that are useful in the control-flow perspective and in the organizational perspective of process mining. The first algorithm counts direct transitions between tasks (the basis for a control-flow model), and the second algorithm counts joint cases between users (the basis for the working together metric [5]). Both are explained in more detail below.

The first algorithm can also be used to count direct transitions between users, which is the basis for the handover of work metric [4]. We will describe this variant only briefly since the algorithm is essentially the same.

3.1 The *flow* algorithm

The purpose of the flow algorithm is to count the transitions between consecutive tasks within each case. For example, if we look at the three cases in the event log of Table 1, we find that there are three occurrences of (a, b), two occurrences of (b, d), one occurrence of (b, c), two occurrences of (e, f), etc.

Let $T = \{a_1, a_2, ..., a_{|T|}\}$ be the set of distinct tasks that appear in the event log, and let (a_i, a_j) denote a transition between two tasks that appear consecutively in the same case id. A transition counting is defined as a function $f : T \times T \rightarrow \mathbb{N}_0$ which gives the number of times that each transition has been observed in the event log. The flow algorithm finds all the values for this function.

For convenience, these values can be stored in a matrix F of size $|T|^2$, which is initialized with zeros. Every possible transition in the form (a_i, a_j) has a value in this matrix, which can be found at row i and column j. Algorithm 1 goes through the event log, and every time a transition (a_i, a_j) is observed, it increments the value at position (i, j) in the matrix.

Algorithm 1 Flow

```
1: Let F be a matrix of size |T|^2
2: Initialize F_ij ← 0 for every i, j
3: for each case id in the event log do
4:     for each transition (a_i, a_j) in the case id do
5:         F_ij ← F_ij + 1
```

3.2 The *handover* algorithm

Let $U = \{u_1, u_2, ..., u_{|U|}\}$ be the set of distinct users that appear in the event log, and let (u_i, u_j) denote a transition between two users that appear consecutively in the same case id. Then substituting T by U and (a_i, a_j) by (u_i, u_j) in Algorithm 1 yields the handover of work matrix H.

3.3 The *together* algorithm

Working together is a metric to extract a social network from an event log. The goal is to find, for each pair of users, how many cases those users have worked together in. For example, in Table 1 it is possible to see that u_1 and u_2 have worked together in all three cases, u_1 and u_3 have worked together in two cases, u_1 and u_4 have worked together only once, etc. The together algorithm calculates this count for every pair of users in the event log.

As in the previous algorithms, this count can be stored in a matrix W of size $|U|^2$, which is initialized with zeros. The values in this matrix are incremented as the algorithm goes through the event log. However, these increments require a little bit more work than in the previous algorithms. Specifically, the algorithm needs to collect the set of users who participate in a case id, and then increment the edge count *for every pair of users in that set*.

Algorithm 2 shows how this can be done. Each pair of users can be denoted as (u_i, u_j). Since (u_i, u_j) and (u_j, u_i) refer to the same pair, the algorithm needs to consider only those pairs in the form (u_i, u_j) with $j > i$. As a result, W will be a triangular matrix.

Algorithm 2 Together

1: Let W be a matrix of size $|U|^2$
2: Initialize $W_{ij} \leftarrow 0$ for every i, j
3: **for** each case id in the event log **do**
4:     Let S be a set of users, initialize $S \leftarrow \emptyset$
5:     **for** each user u_i in the case id **do**
6:         $S \leftarrow S \cup \{u_i\}$
7:     **for** each user $u_i \in S$ **do**
8:         **for** each user $u_j \in S$ such that $j > i$ **do**
9:             $W_{ij} \leftarrow W_{ij} + 1$

4 Parallelization on the CPU

The parallelization on a multi-core CPU is based on the idea of dividing work across a number of threads. According to the description of Algorithms 1 and 2 above, the most natural division of work is by case id, where each thread receives a subset of case ids for processing.

For this processing to be as much independent as possible between threads, each thread will have a local matrix to count the transitions observed in its own subset of case ids. Then, at the end of each thread, it will be necessary to bring these local counts together into a common, global matrix, which stores the combined results of all threads, as illustrated in Figure 2.

Since all threads will be updating the global matrix concurrently, it is necessary to employ thread synchronization to avoid race conditions. In this work, we use a mutex lock on the global matrix to ensure that it is updated by one thread at a time. This effectively reduces parallelism in that section of the code. However, if a thread has a lot of case ids to process, in principle the impact should be reduced, because the time spent on updating the global matrix becomes much shorter than the time spent on processing the case ids.

Figure 2 illustrates the parallelization of the flow algorithm in particular. The parallelization of the together algorithm follows the same approach, with

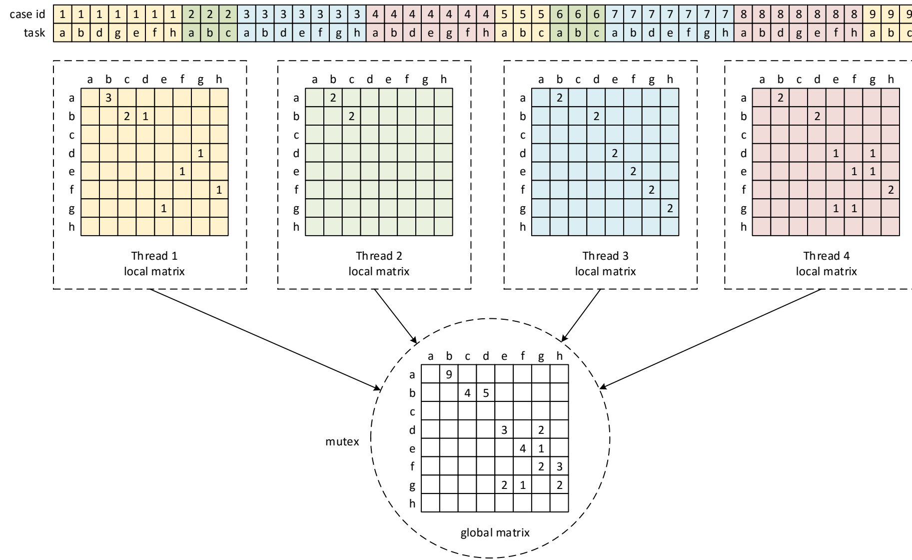

Fig. 2. Multi-threaded parallelization of the flow algorithm on the CPU

the difference being that it works on users rather than tasks, and the calculation of the transition counts is slightly different (see Algorithm 2 vs. Algorithm 1). In any case, at the end of each thread it is necessary to update the global matrix with the transition counts stored in the local matrix.

4.1 Increasing the number of threads

For illustration purposes, Figure 2 shows only four threads but, naturally, this can be extended to an arbitrary number of threads. Increasing the number of threads brings more parallelism, but also more concurrent updates to the global matrix. Since these updates are controlled via a mutex lock, having too many threads may lead to a longer waiting time for the mutex to be released and, eventually, to a decrease in overall performance.

To investigate how far the number of threads can be increased, we carried out an experiment on a machine with two Intel Xeon E5-2630v3 CPUs @ 2.4 GHz with 8 physical cores each, for a total of 16 physical cores. Hyper-threading was enabled [14], so the operating system (Ubuntu) saw 32 virtual cores.

Figure 3 shows the results obtained when running the three algorithms on an event log with 10^6 cases. The dashed line indicates the run-time of the single-threaded version, and the solid line the run-time of the multi-threaded version, as the number of threads is increased. All run-times were averaged over 100 runs of the same algorithm with the same number of threads.

The results in this and other similar experiments indicate that, for the flow and handover algorithms, the ideal number of threads is close to (or even a bit less than) the number of physical cores available in the machine. Any increase

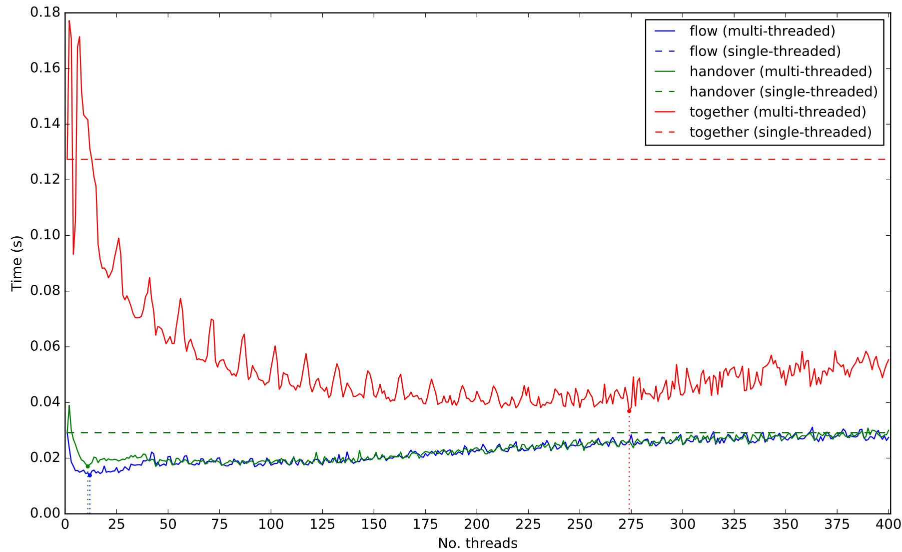

Fig. 3. Run-times of the multi-threaded versions with increasing number of threads

beyond this point results in a decrease in performance. On the other hand, for the together algorithm, there seems to be a benefit in overloading the CPU with a lot of threads. In this algorithm, there is more work to be done for each case id, so the impact of thread synchronization is lower, and the number of threads is allowed to increase up to several times the number of cores.

4.2 Increasing the log size

In another experiment, we increased the log size to investigate its impact on the performance of the multi-threaded versions. For this purpose, the number of cases in the event log was increased in powers of 10, from 10^1 to 10^7 cases (resulting in a log file of 1.8 GB). The number of threads was kept at 10 for the flow and handover algorithms, and at 250 for the together algorithm.

Figure 4 shows a plot of the resulting run-times. Again, the dashed line indicates the run-time of the single-threaded version, and the solid line the run-time of the multi-threaded version, as the number of cases is increased.

The single-threaded versions of the flow and handover algorithms have the same run-times (the two dashed lines are coincident), but in their multi-threaded versions the flow algorithm is slightly faster, because the flow matrix is sparser than the handover matrix.

As for the together algorithm, its run-time drops significantly in the multi-threaded version. For 10^7 cases, the performance gain was $4.0\times$ and it was still growing as the number of cases was being increased. For comparison, the flow and handover algorithms achieved a maximum performance gain of $2.3\times$ and $1.7\times$, respectively, when compared to their single-threaded versions.

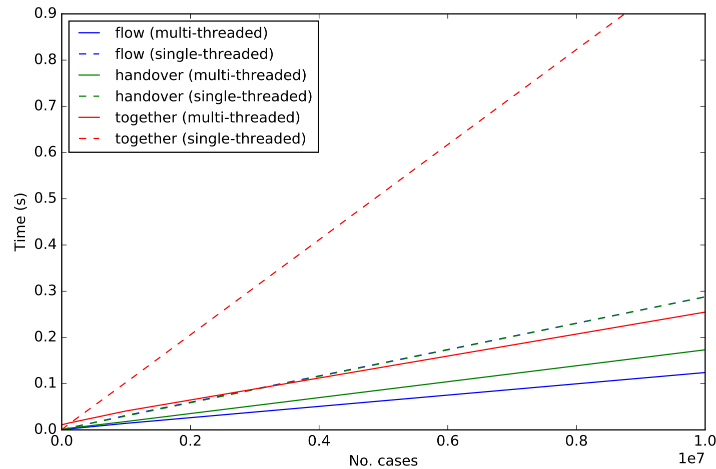

Fig. 4. Run-times of the multi-threaded versions with increasing number of cases

5 Parallelization on the GPU

The parallelization on the GPU requires a completely different approach, because it must match the hardware architecture of modern GPUs. Currently, GPUs have hundreds or even thousands of cores that operate in a thread-synchronous way, with every core executing the same instruction at the same time, but on a different piece of data. This paradigm is usually referred to in the literature as *single instruction, multiple data* (SIMD) [6]. Despite having, in general, a lower clock speed than CPUs, GPUs have so many cores that they can largely outperform the CPU in certain parallelizable tasks.

The Compute Unified Device Architecture (CUDA) [10] is a technology introduced by NVIDIA to make the parallel computing capabilities of GPUs accessible for general-purpose programming. In CUDA, each thread takes care of a single piece of data, and it finishes as soon as that work is done. The idea is to have as many short-lived threads as possible in order to distribute them across the large number of cores available in the GPU.

The work to be done by each thread is programmed into a special function called a *kernel* [15]. The kernel executes on the GPU and is replicated into as many threads as necessary in order to handle all the data that is to be processed. The data must be stored in GPU memory as input and output arrays. Typically, each thread works on a different element (or elements) of these arrays. Complex processing can be done with multiple kernels, where the output array of one kernel becomes an input array to the next, as we will see below.

5.1 Parallelization of the flow algorithm

The GPU parallelization of the flow algorithm is based on the idea of having each thread analyzing a single transition between two consecutive tasks. All transitions will be collected in parallel and at once for the whole event log. In this scenario, each thread must check if the two consecutive tasks being considered actually belong to the same case id. If this is not the case, then the thread writes a special value to the output array, meaning "no transition".

Figure 5 illustrates this step and also what happens afterwards. Once the transitions have been collected, they are sorted and then counted. Both the sorting and the counting are performed on the GPU, with the help of the Thrust library [16]. The sort() routine of the Thrust library makes an efficient use of the GPU to sort large arrays in GPU memory.

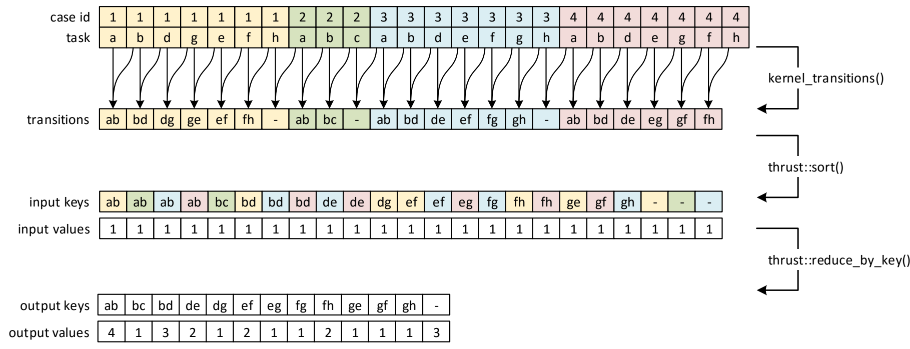

Fig. 5. Parallelization of the flow algorithm on the GPU

After sorting, the counting of transitions can be achieved through a parallel reduction algorithm [10], namely the reduce_by_key() routine available in the Thrust library. For this purpose, it is necessary to have an array with input keys, and another array with input values. Basically, the reduce_by_key() routine sums the values that correspond to the same key. Since the keys are the transitions and the values are all 1, the sum by key yields the transition counts.

Parallelization of the handover algorithm follows the same strategy, with the difference being that threads analyze the transitions between users rather than tasks. The sorting and counting are performed in exactly the same way.

5.2 Parallelization of the together algorithm

The together algorithm is more challenging to parallelize on the GPU because of the need to find the set of distinct users for each case id, and also the need to consider all possible pairs of those users, in the form (u_i, u_j) with $j > i$. We do this in two separated stages, with two different kernels.

As illustrated in Figure 6, the first kernel marks the participants of each case in an output array. For each case id, there is a user mask (corresponding to the

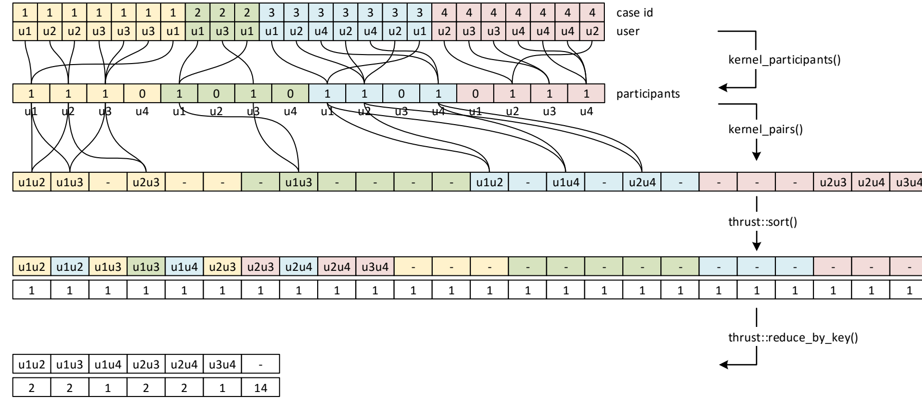

Fig. 6. Parallelization of the together algorithm on the GPU

set of all users U) that is initialized with zeros. The first kernel is launched with one thread per event, to analyze the case id and user of each event. If the thread sees a case id k and a user u_i, it writes 1 at the position $k|U|+i$ in the output array, to mark that u_i participates in case id k.

If the same user appears multiple times in a case id, there will be multiple threads writing 1 at the same position in the output array. However, this does not pose a problem since the operation is idempotent (multiple writes of 1 do not change the result). Also, in Figure 6 we assume that there are only four users (i.e. $|U|=4$) to keep the figure under a manageable size.

The second kernel has a thread for each pair of users in the form (u_i, u_j) with $j > i$. There are $|U|(|U|-1)/2$ such pairs for each case id. If the mask is 1 for both users, then the thread writes the pair $u_i u_j$ to an output array. This pair is written to the position $k|U|(|U|-1)/2 + i|U|+j$ in the output array.

After this, the pairs are sorted and counted as before.

5.3 Results and performance gains

To assess the performance of these GPU parallelizations, we carried out an experiment with a GeForce GTX Titan X with 3072 cores @ 1.08 GHz. Again, we increased the number of cases in powers of 10, from 10^1 to 10^7 cases. Table 2 shows the results, and compares them to the results obtained earlier with the CPU parallelizations in the experiment of Section 4.2.

In Table 2, t_{CPU}, t_{CPU^*} and t_{GPU} denote the run-times of the single-threaded CPU version, multi-threaded CPU version, and GPU version, respectively.

From these results, it becomes apparent that the parallelizations provide a performance gain for events logs with 10^5 cases (around 6×10^5 events) or more. Also, the performance gain of the GPU version is noticeably higher than the multi-threaded CPU version, especially for the flow and handover algorithms.

These measurements were made assuming that all data are already in GPU memory, as is common in practice. If memory transfers between CPU and GPU

No. cases	Algorithm	t_{CPU}	t_{CPU^*}	$t_{\text{CPU}}/t_{\text{CPU}^*}$	t_{GPU}	$t_{\text{CPU}}/t_{\text{GPU}}$
10	flow	0.000001	0.000359	0.003×	0.000437	0.002×
10	handover	0.000001	0.000312	0.003×	0.000439	0.002×
10	together	0.000008	0.000330	0.024×	0.000443	0.018×
100	flow	0.000005	0.000280	0.018×	0.000440	0.011×
100	handover	0.000007	0.000307	0.023×	0.000442	0.016×
100	together	0.000022	0.003294	0.007×	0.000457	0.048×
1000	flow	0.000070	0.000308	0.227×	0.000357	0.196×
1000	handover	0.000058	0.000306	0.190×	0.000364	0.159×
1000	together	0.000190	0.010535	0.018×	0.000516	0.368×
10 000	flow	0.000439	0.000491	0.894×	0.000671	0.654×
10 000	handover	0.000431	0.000647	0.666×	0.000676	0.638×
10 000	together	0.001430	0.011355	0.126×	0.000757	1.889×
100 000	flow	0.003819	0.001957	1.951×	0.000942	4.054×
100 000	handover	0.003827	0.003446	1.111×	0.000948	4.037×
100 000	together	0.011601	0.014514	0.799×	0.002455	4.725×
1 000 000	flow	0.031203	0.014213	2.195×	0.004015	7.772×
1 000 000	handover	0.030080	0.018111	1.661×	0.004028	7.468×
1 000 000	together	0.102830	0.040930	2.512×	0.020586	4.995×
10 000 000	flow	0.287948	0.124074	2.321×	0.034346	8.384×
10 000 000	handover	0.287560	0.173223	1.660×	0.034632	8.303×
10 000 000	together	1.028615	0.254814	4.037×	0.198020	5.195×

Table 2. Run-times and performance gains of CPU and GPU parallelizations[2]

are considered, then the performance gain of the GPU version drops down to roughly the same level as the multi-threaded version.

5.4 BPI Challenge 2016 event logs

The largest event log in the BPI Challenge 2016 – the click-data for non-logged in customers[3] – has about 9.3×10^6 events. For testing purposes, we used the SessionID as case id, and the PAGE_NAME both as task and as user. It should be noted that there are 1381 distinct page names ($|T| = |U| = 1381$), so the transition matrix is much larger than in our previous experiments.

When running the flow algorithm, the multi-threaded CPU version provides little or no gain over the single-threaded version because the transition matrix is very large and takes a long time to be updated by each thread. However, the GPU version provides a consistent performance gain of 7.4×.

When running the together algorithm, the tables are turned. Here, the multi-threaded CPU version provides a performance gain of 5.3×, while the GPU version runs out of memory due to the size of the intermediate arrays.

---

[2] The data and source code used to generate these results are available at: `http://web.tecnico.ulisboa.pt/diogo.ferreira/bpi2016/`

[3] `https://data.3tu.nl/repository/uuid:9b99a146-51b5-48df-aa70-288a76c82ec4`

6 Conclusion

Process mining relies on transition counting procedures which can be accelerated through parallel computing. Parallelization on the CPU provides limited gains due to the need for thread synchronization and merging at some point. Parallelization on the GPU follows a different strategy which avoids thread synchronization and provides higher performance gains, but is limited by GPU memory and memory transfers between CPU and GPU. In any case, both CPU and GPU parallelization provide a promising avenue for accelerating some of the essential tasks that are common to several process mining techniques.

References

1. van der Aalst, W.M.P., Weijters, A.J.M.M., Maruster, L.: Workflow mining: Discovering process models from event logs. IEEE Transactions on Knowledge and Data Engineering **16** (2004) 1128–1142
2. Weijters, A.J.M.M., van der Aalst, W.M.P., de Medeiros, A.K.A.: Process mining with the HeuristicsMiner algorithm. Technical Report WP 166, Eindhoven University of Technology (2006)
3. Günther, C.W., Rozinat, A.: Disco: Discover your processes. In: BPM 2012 Demonstration Track. Volume 940 of CEUR Workshop Proceedings. (2012)
4. van der Aalst, W.M.P., Song, M.: Mining social networks: Uncovering interaction patterns in business processes. In: Business Process Management. Volume 3080 of LNCS., Springer (2004) 244–260
5. van der Aalst, P.W.M., Reijers, A.H., Song, M.: Discovering social networks from event logs. Computer Supported Cooperative Work **14**(6) (2005) 549–593
6. Rauber, T., Rünger, G.: Parallel Programming for Multicore and Cluster Systems. 2nd edn. Springer (2013)
7. Veiga, G.M., Ferreira, D.R.: Understanding spaghetti models with sequence clustering for ProM. In: Business Process Management Workshops. Volume 43 of LNBIP., Springer (2010) 92–103
8. Kundra, D., Juneja, P., Sureka, A.: Vidushi: Parallel implementation of alpha-miner algorithm and performance analysis on CPU and GPU architecture. In: Business Process Management Workshops. Volume 256 of LNBIP., Springer (2016)
9. Butenhof, D.R.: Programming with POSIX Threads. Addison-Wesley (1997)
10. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with CUDA. ACM Queue **6**(2) (March/April 2008) 40–53
11. Ferreira, D.R., Vasilyev, E.: Using logical decision trees to discover the cause of process delays from event logs. Computers in Industry **70** (June 2015) 194–207
12. van Dongen, B.F., van Der Aalst, W.M.P.: A meta model for process mining data. In: EMOI-INTEROP'05. Volume 160 of CEUR Workshop Proceedings. (2005)
13. Verbeek, H.M.W., Buijs, J.C.A.M., van Dongen, B.F., van der Aalst, W.M.P.: XES, XESame, and ProM 6. In: Information Systems Evolution. Volume 72 of LNBIP., Springer (2011) 60–75
14. Magro, W., Petersen, P., Shah, S.: Hyper-threading technology: Impact on compute-intensive workloads. Intel Technology Journal **6**(1) (2002)
15. Nickolls, J., Dally, W.J.: The GPU computing era. IEEE micro **30**(2) (2010) 56–69
16. Bell, N., Hoberock, J.: Thrust: a productivity-oriented library for CUDA. In: GPU Computing Gems: Jade Edition. Morgan Kaufmann (2011) 359–371