

Multi-threaded approach in generating frequent itemset of Apriori algorithm based on trie data structure

Ade Hodijah, Urip Teguh Setijohatmo, Ghifari Munawar, Gita Suciana Ramadhanty

Informatics and Computer Engineering, Politeknik Negeri Bandung, Bandung, Indonesia

Article Info

Article history:

Received Jul 31, 2021

Revised Jun 30, 2022

Accepted Jul 08, 2022

Keywords:

Apriori algorithm

Flynn's taxonomy

Indexing

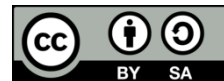
Multi-threaded

Trie data structure

ABSTRACT

This research is about the application of multi-threaded and trie data structures to the support calculation problem in the Apriori algorithm. The support calculation results can search the association rule for market basket analysis problems. The support calculation process is a bottleneck process and can cause delays in the following process. This work observed five multi-threaded models based on Flynn's taxonomy, which are single process, multiple data (SPMD), multiple process, single data (MPSD), multiple process, multiple data (MPMD), double SPMD first variant, and double SPMD second variant to shorten the processing time of the support calculation. In addition to the processing time, this work also considers the time difference between each multi-threaded model when the number of item variants increases. The time obtained from the experiment shows that the multi-threaded model that applies a double SPMD variant structure can perform almost three times faster than the multi-threaded model that applies the SPMD structure, MPMD structure, and combination of MPMD and SPMD based on the time difference of 5-itemsets and 10-itemsets experimental result.

This is an open access article under the [CC BY-SA](https://creativecommons.org/licenses/by-sa/4.0/) license.



Corresponding Author:

Ade Hodijah

Informatics and Computer Engineering, Politeknik Negeri Bandung

Bandung, Indonesia

Email: adehodijah@jtk.polban.ac.id

1. INTRODUCTION

Many algorithms can be used in association rule mining. The commonly used algorithm for association rule mining in transactional databases is the Apriori algorithm [1]-[3]. Bhandari *et al.* [4] presents a new approach to data separation by modifying the native Apriori algorithm using a tree-based approach. The study shows a method that helps find itemsets that occur frequently but with fewer tables. So the amount of space required to store the table is significantly reduced.

The hash tree [5]-[7] has been implemented in the distributed computing environment of Spark. Another used Hadoop-map reduce framework [8], [9], and graph computing techniques for frequent itemsets mining have been published by Zhang *et al.* [10]. However, in this research, the trie-based of Apriori algorithm was implemented on a single node.

In the problem of market basket analysis, the data that must be processed is vast. Calculating support count in the generation of candidate sets is a bottleneck process. It will take a long time to use only a basic Apriori algorithm with a trie data structure. Therefore, the multi-threaded concept is tried to be applied whether it can be utilized to minimize the time required in processing the support count. To achieve the counting task, the candidate set positions in the trie are calculated by applying a formula to get the address or index of the candidate. In this research, it is called the indexing candidate set.

This research topic is about utilizing multithread to get frequent itemset mining based on [11], in which this paper found that there is still a tiny amount of research using the multi-threaded approach. In [11] discusses the development of research solutions for obtaining frequent itemset mining over the last 25 years. This paper categorizes solutions into sequential, multithreading, and distributed computing. According to Luna *et al.* [11], using multi-threaded solutions with shared memory is not yet so crucial mainly because memory requirements are one of the main problems to be solved in frequent item mining (FIM). It explains why distributed computing-based solutions are increasingly emerging, especially with the big data explosion. Some research works on distributed computing environments are represented in [12], [13]. Both results are experiments the Spark, while in [13], Spark is developed instead of Hadoop because in-memory computation, in [12] initially investigates issues involved with pattern mining approaches and related techniques like Apache Hadoop, Apache Spark, parallel and distributed processing. It then examines significant parallel, distributed, and scalable pattern mining developments, analyzes them from a big data perspective and identifies difficulties in designing algorithms.

Then years later, with increasing interest in graphics processing units (GPUs), several new parallel GPU-based approaches were proposed. One of the significant works of the GPU approach is as written in [14], in which three high-performance computing (HPC)-based versions of single scan have been proposed for frequent itemset mining, namely GPU single scan (GSS-GPU) which implements single scan (SS) on a GPU, cluster single scan (CSS)-cluster single scan, and CGSS-cluster GPU single scan. The results reveal that CGSS outperforms GSS and CSS which implements SS on a cluster for large databases. The results also show that cluster GPU single scan (CGSS) which implements GSS-GPU on a Cluster outperforming the sophisticated HPC-based FIM approach by using large databases for different threshold support values. Another works similar to [14] is GMiner, explained in [15]. It achieves blazing-fast performance by fully utilizing the GPU's computing power. However, this method oppositely performs the mining task. Instead of storing and exploiting patterns in the intermediate level tree, it mines patterns from the first level of the enumeration tree.

Under both works, our research aim is not yet exploiting hardware support as the GPU but establishing the first fundamental process model. Ultimately, some multi-threaded [16] model in this research is used to test how fast the time is needed in calculating the support count by using an Apriori algorithm. The model has been enhanced with a trie data structure and tested on a single node.

2. METHOD

The research conducted is a quantitative research using experimental research techniques. First, experiments were carried out to apply the concept of multi-threaded and trie to the support subset calculation process. After that, analyze the time required to carry out the process quantitatively on the number of threads and variants of certain items.

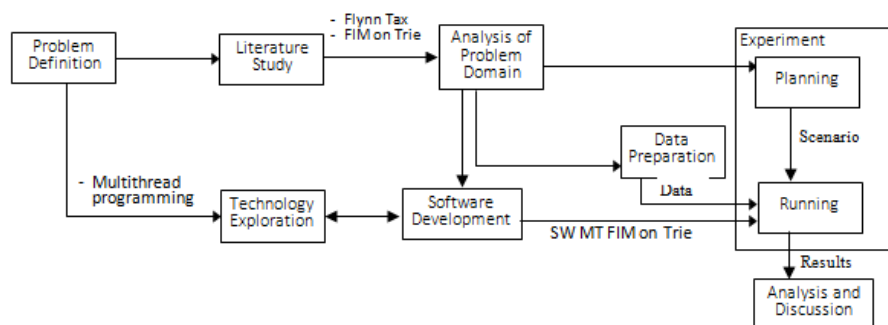


Figure 1. The proposed method for this works

Figure 1 is the methodology of this research which consist of eight steps:

- 1) Problem definition: this step defines the problem that needs to be addressed from potential problems mainly from the history of our work, and one of them is the proposed topic.
- 2) Literature study: the study mainly focuses on extending past work about performance analysis in determining FIM to be further equipped by some means of parallel processing. So this research starts on the thread-level parallelization by utilizing multithread on some steps of processes in a pipelining manner. The other subjects related to parallelization are Flynn taxonomy, inter-process communication, and race condition that cannot be abandoned.

- 3) Technology exploration: along the way to the literature study, technologies related to the subjects mentioned are explored. In this research much of the implementation will use Java as the main platform so the components of logic, for example, multithread programming and mutual exclusion mechanism will be implemented using Java.
- 4) Analysis of problem domains: the analysis was conducted to determine the behavior of the support calculation process and the index search method based on the analyzed support calculation pattern and the data structure of the trie needed to store the support value. The index search method is required to find the index location of a subset of the trie.
- 5) Data preparation: this step is conducted after the domain problem's behavior is understood and specifically defined and modeled. The data is a dummy, and the data format is the text that resembles transactions and will be created using an automated random algorithm.
- 6) Software development: the development will use a multi-threaded model adopted from computer architecture categories [17]-[20], namely Flynn taxonomy. Since the model is at the process level, the term process will be used instead of instruction. So multiple instructions multiple data (MIMD) becomes multiple process multiple data (MPMD). Among the processes on the models are then chained by pipelining approach [21]. Despite the single process, single data (SPSD), are single process, multiple data (SPMD), multiple process, single data (MPSD), and MPMD, this work restricts just two models, MPMD and SPMD, since our focus is on multi-threaded separated data (MD-multiple data) to achieve more parallel works.
- 7) Planning of experiment: this step will prepare the running of an experiment by creating scenarios reflecting regions to be observed related to the research question. The scenarios are generated based on the analysis result so that the target direction will be well reserved.
- 8) Execution of experiments.

Experiments were carried out for each multi-threaded model based on the scenarios. The processing time of each model is compared to analyze the speed efficiency. The number of transactions that are the subject of the experiment is 100,000 transactions with variance items are five and 10 items. The dataset used for the experiment is a dataset that has been randomly generated with a different number of item variants from each record in the dataset. Table 1 contains an example of the data generated for the experiment's needs with the number of item variants of 10.

Table 1. The example of dataset

Transaction ID	Items
T1	{3, 5, 10}
T2	{1, 2, 3, 4, 5, 6, 7, 8, 10}
T3	{10}

The run time obtained from the experimental results of each model determines the performance [22]. More specifically, the quality of each multi-threaded model is based on the following factors:

- Average processing time for each model.
- The difference in the period when the number of item variants increases in each model.

Time quality is declared to be better if one of the multi-threaded models is applied faster than the other.

3. RESULTS AND DISCUSSION

The multi-threaded process is implemented in read, combine, index, and support process. In contrast, the frequent process is excluded, as seen in Figure 2. First, the data structure used to represent trie is a two-dimension array. The row of the array is $2^n - 1$ since it represents the number of a subset of superset of all possible combinations of n items. If n , the number of item, is 3 then the superset is $\{\{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$. Each of the subsets is called the candidate set, and it is required that each candidate set will have support reflecting the popularity of the sale items. It means support is calculated based on the transactions of the item. Each subset's location, which we named the index, is fixed at a certain position on the array and can be reached by calculating a formula instead of a one-by-one exhausting search. The formula is a function that gets the subset or superset of items sold in each transaction as an input parameter and returns an integer as an index of the array where the subset is located. If transaction $T1$ consist of item 2 and 3 then each of support $[\text{hash}(\{2\})]$, support $[\text{hash}(\{3\})]$, and support $[\text{hash}(\{2,3\})]$ will be incremented by 1. From the above description, we will have six processes as:

- 1) Read n the number of item sales, create the array of $2^n - 1$ rows

- 2) Read the transaction data (read)
- 3) Generate superset (combine) of item set on transaction of step 2
- 4) Find index using Hash-node calculation (index)
- 5) Update support count on trie (support)
- 6) Get a frequent itemset based on a minimum threshold (frequent)

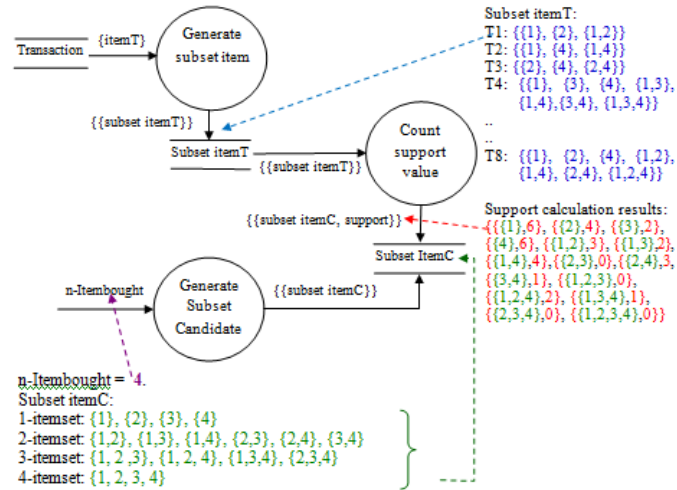


Figure 2. Support value calculation process

One of the essential tasks in this research is the formula, the index calculation method. The index search method used in this study is based on the pattern of each subset length, the number of branch points found, and the number of subsets of one branch point using the trie [23] data structure. Figure 3 illustrates the modified Hash-node calculation trie tree of this research.

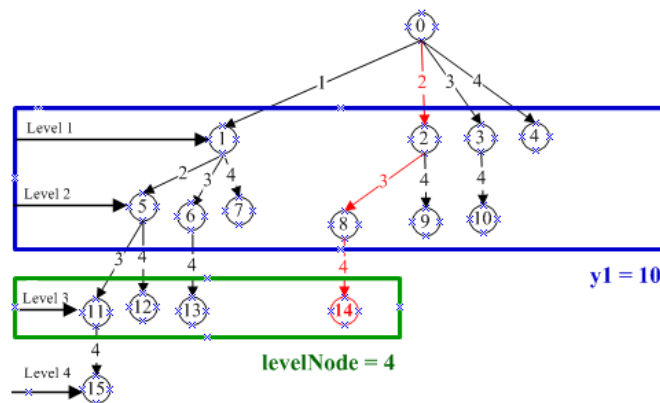


Figure 3. Example for index of itemset {2, 3, 4} is 14, which are $y_1 = 10$ and $levelNode = 4$

There are four steps to get the index of the k-itemset as:

- Step 1. y_1 : get the number of subsets by using the y_1 [23]. Itemset of y_1 are: $\{\{1\}, \{2\}, \{3\}, \{4\}, \{1,2\}, \{1,3\}, \{1,4\}, \{2,3\}, \{2,4\}, \{3,4\}\}$.

$$y_1 = \sum_{k=1}^{len-1} \frac{n!}{k!(n-k)!} \tag{1}$$

The y_1 of Hash-node calculation [23] is still used, but the y_2, y_3, y_4, y_5 via branch point (BP) calculation has been replaced with the algorithm of searching the node based on the generated itemset combination using powerset algorithm approach [24] at step 2. The algorithm is a development of the y_1 as:

$$y_1' = \sum_{k=level}^{level} \frac{n!}{k!(n-k)!} \tag{2}$$

The y_1' cannot get the location for a given itemset, but it returns the number of all nodes in the trie tree at a given level. It still needs further calculations that return the value of the location by checking whether the combination of itemset returned is the same as the itemset being searched for. Therefore, the algorithm does not perform the generate itemset combination from a 1-itemset. Still, it is carried out only on the combined value for a certain level length of k until the itemset combination (node) which is looking for is found. A sample pseudo-code for the modified Hash-node calculation process is as:

- Step 2. *levelNode*: get all nodes of the itemset by tracing the trie at a certain level of k by calling the combination method, where size is k or a level in the trie tree (see Figure 3). All Itemsets of *levelNode* are: {{1, 2, 3}, {1, 2, 4}, {1, 3, 4}, {2, 3, 4}}. Figure 4 shows the main program of *levelNode* itemset generation and Figure 5 shows the module for combination algorithm generation at a certain level of trie tree.

```
for (int level=k; level<=k; level++){
    powerSet.addAll(combination(Arrays.asList(itemset),level));
}
```

Figure 4. The main program, which calls the combination module

```
<T>List<List<T>> combination(List<T> values,int size){
    if (0 == size) {
        return Collections.singletonList
            (Collections.<T> emptyList());
    }
    if (values.isEmpty()) {
        return Collections.emptyList();
    }
    List<List<T>> combination = new LinkedList<List<T>>();
    T actual = values.iterator().next();
    List<T> subSet = new LinkedList<T>(values);
    subSet.remove(actual);
    List<List<T>> subSetCombination = combination(subSet,size-1);
    for (List<T> set : subSetCombination) {
        List<T> newSet = new LinkedList<T>(set);
        newSet.add(0, actual);
        combination.add(newSet);
    }
    combination.addAll(combination(subSet, size));
    return combination;
}
```

Figure 5. Module to get list of *levelNode* combination, where $values = Arrays.asList(itemset)$ and $size = k$ [25]

To get a particular index, it needs to check the similarity between the itemset and the result of *levelNode* list.

- Step 3. *levelNode'*: do itemset similarity checking. If the itemset being searched for is {1, 2, 4} then the values of *levelNode'* are: {{1, 2, 3}, {1, 2, 4}}, as seen in Figure 6.
- Step 4. *index*: get the index location of a subset at the trie tree by summing the value of y_1 and *levelNode*.

$$index = y_1 + levelNode' \quad (3)$$

```
List<List<Integer>> LevelNode(List<List<Integer>> powerSet,
    Integer[] itemset){
    List<List<Integer>> result = new LinkedList<List<Integer>>();
    for (int i = 0; i < powerSet.size(); i++) {
        List<Integer> item = powerSet.get(i);
        result.add(item);
        for (int j = 0; j < item.size(); j++) {
            boolean isFound = false;
            if(item.get(j) == itemset[j]){
                isFound = true;
                if(j+1 == item.size()) {
                    i = powerSet.size();
                }
            }
            if(!isFound) {
                j = item.size();
            }
        }
    }
    return result;
}
```

Figure 6. Module to get the list of *levelNode* until it is the same with itemset being searched for

Two main thread models are applied in this research: lightweight process (LWP) and big process (BP). Here's the process behavior for the LWP, as shown in Figure 7. The BP thread model is shown in Figure 8.

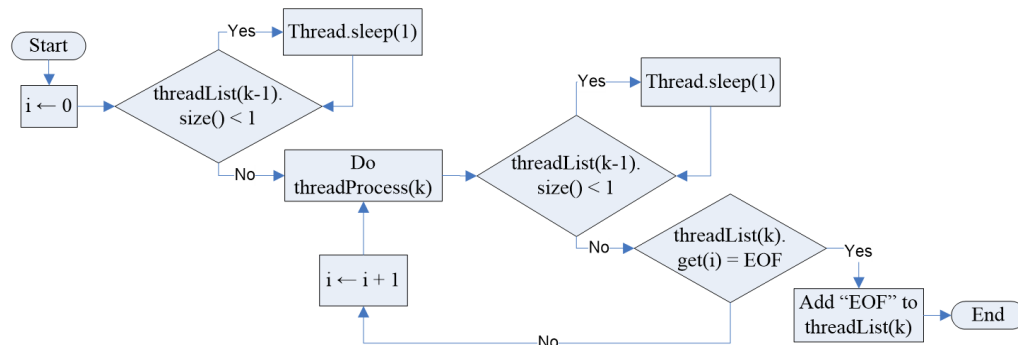


Figure 7. LWP thread model

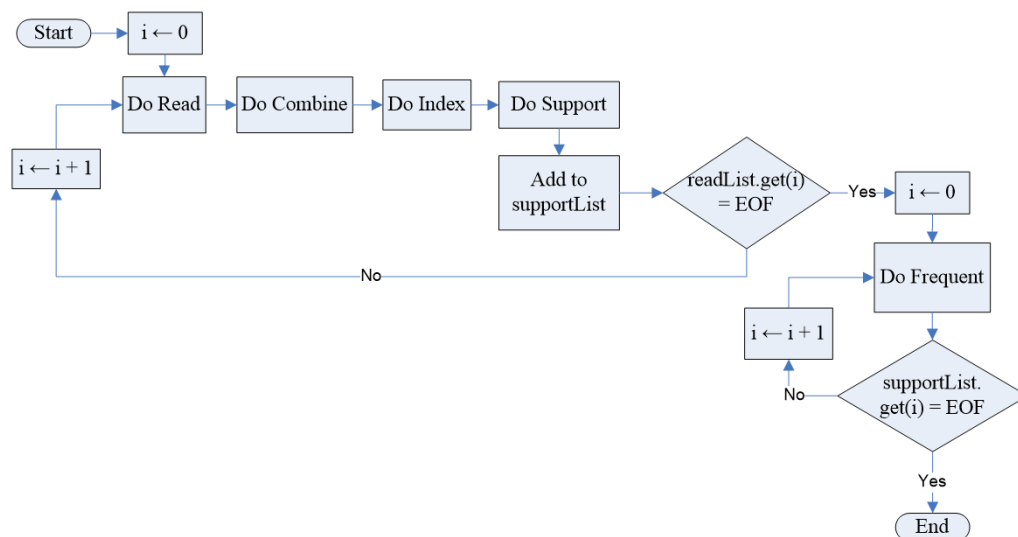


Figure 8. BP thread model

All threads for the LWP model use the same model overall, except for the read thread, because the data has already been prepared. Furthermore, the Frequent process is excluded from the main thread cycle. Still, the Frequent process is executed in a class of the main program when all (read, combine, index, and support) of the calculating process of the support value of each candidate set have been done.

There are two types in LWP, which are threadProcess and threadList. The threadProcess executes read, combine, index, and support for new transaction data, while the threadList saves the result of the previous read, combine, index, and support process. So, for example, if the object for thread two is looking for a subset of transactions, then the object needed is a threadList, and the process executed is threadProcess. After the threading process is complete, the last object element marker (EOF) is added that the processed object has been completed.

Before the main process loop (loop 2) starts, the object size is checked in loop 1. If the object for the process is still empty, then the thread goes to a sleep state. Likewise, loop two does not run if the object size is still empty, but after the main process (eg. threadProcess) is successfully carried out once using a repeat until () control block and so on until the contents of the object are the markers for the last object element of the process in the previous thread.

The thread model with double-checking on the object size is applied that way because if you take the object size directly, the result may be less than the actual object size, i.e. the object size as a result of processing from the previous thread. It is caused by the kernel's erratic running of thread sequences. Process behavior for BP model threads is different from LWP, where threads are applied to the size of the dataset divided by a certain number of sizes, not to each process like the thread model in LWP.

All threads for the BP model use the same model as the process behavior on a single thread [23], in which the order of processes is maintained. There are two main parts of the process, namely the first to process read, combine, index, and support for $dataset[i]$ to EOF of transaction data. After the support values are updated for all candidate sets in the trie, then the frequent process starts from the first index of the support list to the last index of it, where a node in the trie will be removed for the index which has support value is less than the threshold value (minimum support).

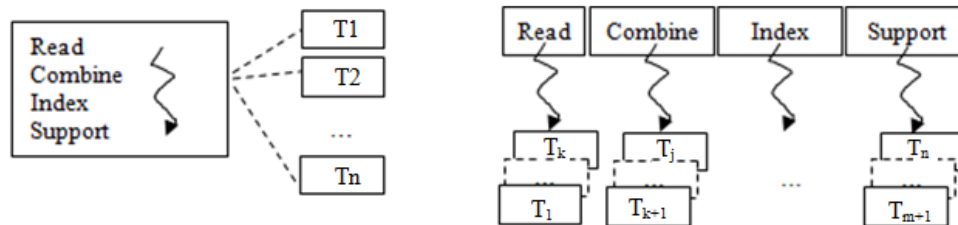


Figure 9. Read, combine, index, and support process on BP and LWP

Based on the working principle of threads from BP and LWP, as seen in Figure 9, the following multi-threaded model was developed in this study by utilizing the working principle of user threads. Where developers create threads to handle multiple control flows in software using the API provided by the thread library [26]. Here's the main program (simplified) for each multi-threaded model based on Flynn's taxonomy approach [17].

- 1) SPMD use data-driven approach: one thread contains five stages of the support search process for a certain amount of data. The total transaction data is divided into four threads, as seen in Figure 10. There are three parts of the program, namely: 1) declaration of thread object; 2) thread for read, combine, index, and support process; and 3) frequentprocess, as seen in Figure 10. Part 1 and 3 of the program are always the same for all thread models, so the program discussion for the next will only focus on part 2. The four stages of the process start from the read, combine, index, support, and the last is the frequent process which is executed in a class of the main program or excluded from the multi-threaded process. The number of transaction data used as the test dataset is 100,000, so T1 performs the calculation process for the support value of each candidate set for transaction data starting from 0 to 25,000 transaction data and so on until T4 starts from 75,000 to 100,000 transaction data. The total thread in this model is five, which are the main program and four threads mentioned.

```

1  MainThread threadObject = new MainThread(N itemsetCombination);
   Thread T1 = new Thread(new ReadCombineIndexSupport(threadObject, 1));
   Thread T2 = new Thread(new ReadCombineIndexSupport(threadObject, 2));
2  Thread T3 = new Thread(new ReadCombineIndexSupport(threadObject, 3));
   Thread T4 = new Thread(new ReadCombineIndexSupport(threadObject, 4));
   T1.start(); T2.start(); T3.start(); T4.start();
   try
   {
       T1.join(); T2.join(); T3.join(); T4.join();
   }
   catch(Exception e) { System.out.println("Interrupted"); }
3  synchronized (threadObject) {
       threadObject.setListFrequent();
   }

```

Figure 10. SPMD multi-threaded models

- 2) MPSD use lightweight process (LWP) approach: the multi-threaded MPSD model used in this study can be seen in Figure 11. One thread contains one stage (substep) of finding support for all data. T1 to read all transaction data (read), T2 to find a superset of all transaction data (combine), T3 to find the index of each subset that has been obtained (index), and T4 to add support values to the candidate set with the same index has been obtained (support). At the start, all threads that don't have data sleep first, but not the case for thread reading transactions since the data is available. The thread is then executed and scheduled according to the kernel. After all threads (read, combine, index, and support) have been executed, the last is to do a frequent process executed in the main program. The total thread in this model is five, the main program and four threads mentioned. For MPSD, where processes run using a shared

variable, they are handled with care for the possibility of race conditions, as shown in Figure 12. The shared variables are rtrans, Sset, and ndx. Each variable is used by at least two processes depending on the thread number created for each process. The two competing process works in a producer-consumer pattern, where the counters of the variable content are the shared variables. So when the process role is producer, they will sleep as the shared memory is full as opposed to the one with consumer role in which they will sleep as the memory is empty. Some processes have both roles. This mechanism will affect performance issues.

```
Thread T1 = new Thread(new Read(threadObject));
Thread T2 = new Thread(new Combine(threadObject));
Thread T3 = new Thread(new Index(threadObject));
Thread T4 = new Thread(new Support(threadObject));
```

Figure 11. MPSD multi-threaded models

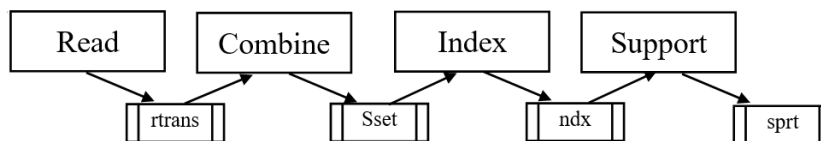


Figure 12. LWP with a pipeline

- 3) MPMD: data is divided into groups, so multiple data are available to be processed separately. The MPMD of this study can be seen in Figure 13. The process behavior of the MPMD model is the same as the previous MPSD model, but one LWP only works on 1/4 of the total transaction data. The LWP threads in this model are 16, which work in parallel [27] which consists of 4 groups of datasets that are divided together with 1 group of datasets done by one LWP cycle, which consists of 4 threads that work in parallel as well, namely read, combine, index, and support threads. So, the total number of threads of this model is 17.

```
Thread T1 = new Thread(new Read(threadObject, 1));
Thread T2 = new Thread(new Read(threadObject, 2));
Thread T3 = new Thread(new Read(threadObject, 3));
Thread T4 = new Thread(new Read(threadObject, 4));
Thread T5 = new Thread(new Combine(threadObject, 1));
Thread T6 = new Thread(new Combine(threadObject, 2));
Thread T7 = new Thread(new Combine(threadObject, 3));
Thread T8 = new Thread(new Combine(threadObject, 4));
Thread T9 = new Thread(new Index(threadObject, 1));
Thread T10 = new Thread(new Index(threadObject, 2));
Thread T11 = new Thread(new Index(threadObject, 3));
Thread T12 = new Thread(new Index(threadObject, 4));
Thread T13 = new Thread(new Support(threadObject, 1));
Thread T14 = new Thread(new Support(threadObject, 2));
Thread T15 = new Thread(new Support(threadObject, 3));
Thread T16 = new Thread(new Support(threadObject, 4));
```

Figure 13. MPMD multi-threaded models

- 4) The first variant of double SPMD: basically, the process behavior of this double SPMD model is the same as the previous SPMD model, but one task is carried out by four threads that work in parallel on 1/4 amount of data. For this scenario, each task, starting from read works in isolation and sequential, one task after another. Each task is handled by four threads that run parallel. No other process is participating, waiting for their turn. After the read task finishes, the task of combine, also done by four threads and gets its turn. Then the total of threads is 17, as shown in Figure 14. The result of each task will be stored separately. For example, the result of T1 will be stored in ReadList (k), where the value of k starts from 1 to 4, according to the value of T1 to T4.


```

Thread T1 = new Thread(new Read(threadObject, 1));
Thread T2 = new Thread(new Read(threadObject, 2));
Thread T3 = new Thread(new Read(threadObject, 3));
Thread T4 = new Thread(new Read(threadObject, 4));
T1.start(); T2.start(); T3.start(); T4.start();
try { T1.join(); T2.join(); T3.join(); T4.join(); }
catch(Exception e) { System.out.println("Interrupted"); }
Thread T5 = new Thread(new Combine(threadObject, 1));
Thread T6 = new Thread(new Combine(threadObject, 2));
Thread T7 = new Thread(new Combine(threadObject, 3));
Thread T8 = new Thread(new Combine(threadObject, 4));
T5.start(); T6.start(); T7.start(); T8.start();
try { T5.join(); T6.join(); T7.join(); T8.join(); }
catch(Exception e) { System.out.println("Interrupted"); }
Thread T9 = new Thread(new Index(threadObject, 1));
Thread T10 = new Thread(new Index(threadObject, 2));
Thread T11 = new Thread(new Index(threadObject, 3));
Thread T12 = new Thread(new Index(threadObject, 4));
T9.start(); T10.start(); T11.start(); T12.start();
try { T9.join(); T10.join(); T11.join(); T12.join(); }
catch(Exception e) { System.out.println("Interrupted"); }
Thread T13 = new Thread(new Support(threadObject, 1));
Thread T14 = new Thread(new Support(threadObject, 2));
Thread T15 = new Thread(new Support(threadObject, 3));
Thread T16 = new Thread(new Support(threadObject, 4));
T13.start(); T14.start(); T15.start(); T16.start();
try { T13.join(); T14.join(); T15.join(); T16.join(); }
catch(Exception e) { System.out.println("Interrupted"); }

```

Figure 14. The first variant of double SPMD multi-threaded models

- 5) The second variant of double SPMD: there is another double SPMD model run by half of the first double SPMD model. The idea of this model is to find what mechanisms are the potential to get a more efficient run time. The first finding is to give more threads to the complex process where more workers may help. The index calculation process used by Hash-node calculation [23] is the one eligible for the qualification. Therefore the number of threads for Index calculation is twice the first variant of the double SPMD model. The second finding is that the level of lightweight of a process is optimum. It seems the index calculation and support can be united together since the support process is too light that it consists of just a statement of incrementing the support value of the index. If they are kept separately, they need to synchronize access to the competing shared variables, which is not the case if they are kept together. The new model is shown in Figure 15.

```

Thread T1 = new Thread(new Read(threadObject, 1));
Thread T2 = new Thread(new Read(threadObject, 2));
Thread T3 = new Thread(new Read(threadObject, 3));
Thread T4 = new Thread(new Read(threadObject, 4));
T1.start(); T2.start(); T3.start(); T4.start();
try { T1.join(); T2.join(); T3.join(); T4.join(); }
catch(Exception e) { System.out.println("Interrupted"); }
Thread T5 = new Thread(new Combine(threadObject, 1));
Thread T6 = new Thread(new Combine(threadObject, 2));
Thread T7 = new Thread(new Combine(threadObject, 3));
Thread T8 = new Thread(new Combine(threadObject, 4));
T5.start(); T6.start(); T7.start(); T8.start();
try { T5.join(); T6.join(); T7.join(); T8.join(); }
catch(Exception e) { System.out.println("Interrupted"); }
Thread T9 = new Thread(new IndexSupport(threadObject, 1));
Thread T10 = new Thread(new IndexSupport(threadObject, 2));
Thread T11 = new Thread(new IndexSupport(threadObject, 3));
Thread T12 = new Thread(new IndexSupport(threadObject, 4));
Thread T13 = new Thread(new IndexSupport(threadObject, 5));
Thread T14 = new Thread(new IndexSupport(threadObject, 6));
Thread T15 = new Thread(new IndexSupport(threadObject, 7));
Thread T16 = new Thread(new IndexSupport(threadObject, 8));
T9.start(); T10.start(); T11.start(); T12.start();
T13.start(); T14.start(); T15.start(); T16.start();
try
{
    T9.join(); T10.join(); T11.join(); T12.join();
    T13.join(); T14.join(); T15.join(); T16.join(); }
catch(Exception e) { System.out.println("Interrupted"); }

```

Figure 15. The second variant of double SPMD multi-threaded models

This work conducted five experiments using processor Intel® Core™ i5-8250U CPU @ 1.60 GHz 1.80 GHz, installed RAM is 12.0 GB, the system type is a 64-bit operating system, x64-based processor, and Java [26] multithreading libraries. The detail of the experimental results is shown in Table 2 for 5-itemset and 10-itemset, where the SPMD model is the fastest for the 5-itemset, while the second double SPMD is the fastest for 10-itemset. Meanwhile, the MPMD model is longer than others, both 5-itemset and 10-itemset. Furthermore, the first and second double SPMD model is faster than others on average.

Table 2. The experimental results for A: SPMD; B: MPSD; C: MPMD; D: the first variant of double SPMD; and E: the second variant of double SPMD

Multi-threaded model	Total threads	Number of item variants	Processing time average (ms)
A	5	5	1494
		10	388813
B	5	5	2553
		10	370323
C	17	5	2838
		10	367635
D	17	5	1688
		10	136386
E	17	5	1608
		10	125601

Table 2 shows the time difference between the longest for 5-itemset is 2838 (MPMD) – 1494 (SPMD) = 1344 ms and for 10-itemset is 388813 (SPMD) – 125601 (second double SPMD) = 263212 ms. Figure 16 shows the time difference based on the number of item variants and thread models. It shows that the double SPMD model is faster than others, with the fastest is the second variant of double SPMD, which is 123993 ms or 123 seconds, and the longest of all multi-threaded models is the SPMD model, 387319 ms or 387 seconds. The time difference for the number of item variants and thread models is 387–123=264 seconds. Furthermore, from the first of three rows, namely SPMD, MPSD, and MPMD, the pipeline structure adds cost to the sleep and wakeup mechanism and needs to be carefully utilized. It happens as the index and support process is united, which affects the performance significantly.

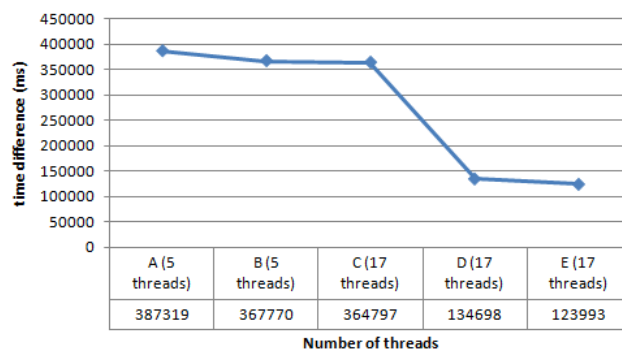


Figure 16. Time difference of 5-itemsets and 10-itemsets for A: SPMD, B: MPSD, C: MPMD, D: the first variant of double SPMD, and E: the second variant of double SPMD

4. CONCLUSION

This research conducted a multithread process by examining five models based on Flynn taxonomy to get insight into how threads work and how they behave inter-process communication to achieve synchronization and their effect on performance. One of the findings from the observation is to have a model process of multithread with optimal synchronization. Because the greater the number of threads is not always the lower the time processing is obtained. Although the average result of the double variant of SPMD (17 threads) is faster than SPMD (5 threads), the MPMD (17 threads) is longer than the double variant of SPMD (17 threads) for both 5-itemset and 10-itemset, besides that the MPMD (17 threads) is longer than SPMD (5 threads) for 5-itemset. Our work has achieved just the beginning of the requirement and needs to address some related subjects. Up to this point, the research had two ideas for reducing synchronization: 1) isolating each process and, at the same time, double the worker; and 2) they were uniting or joining the wrong size of LWP. On average, the SPMD (BP thread) model is faster than MPMD (LWP thread) model. There are more logical algorithms for synchronizing the MPMD model selection process, starting from the combine process, then index process, and the last is the support process. Each process has to guarantee that the list of datasets in (i-1) or the previous process has been completed before the (i) process is finished. The selection is based on the list size between (i-1) and (i) processes. Otherwise, the SPMD model has not had too much selection process because the (i) process will not be started before all of the datasets in the (i-1) process have been completed.

Further research is recommended to take at least two paths. The first path is to learn more about how threads behave, especially on the inter-process communication (IPC) and related blocking mechanisms as the thread is read, sleeps, and wakes up. It's also a potential candidate for trying the child process instead of thread or the combination. The second path is conducted on the same topic in distributed computing environments, such as the Hadoop-map reduce framework or the Spark platform. The index calculation process requires complex steps. It is recommended to break down the index process to become a specific threads process that conducts the calculation of y_1 and *levelNode*. The algorithm's complexity requires a large enough memory to generate the combination of itemset so that only a small amount of heap space is available. This research makes the generation of itemset combination maximum for a 10-itemset. This bit of space will also affect the speed of the following process. The less heap space available, the slower the process runs. It also affects the number of variants of transaction items. Because the greater the number of transaction items, the greater the heap space needed to calculate the subset index. For this reason, it is necessary to try experimenting with distributed computing. It is also recommended to use more extensive data. With the relatively large amount of experimental data, much more scenarios can be done, and possibly more insight can be taken from the results.




REFERENCES

- [1] C. D. Kumar and G. Maragatham, "Analysis of apriori and fp-growth algorithm in a distributed environment," *International Journal of Pure and Applied Mathematics*, vol. 120, no. 6, pp. 2779–2788, 2018. [Online]. Available: <https://acadpubl.eu/hub/2018-120-6/2/200.pdf>
- [2] H. Xie, "Research and Case Analysis of Apriori Algorithm Based on Mining Frequent Item-Sets," *Open Journal of Social Sciences*, vol. 9, no. 4, pp. 458–468, 2021, doi: 10.4236/jss.2021.94034.
- [3] Z. Li, X. Li, R. Tang, and L. Zhang, "Apriori Algorithm for the Data Mining of Global Cyberspace Security Issues for Human Participatory Based on Association Rules," *Frontiers in Psychology*, vol. 11, 2021, doi: 10.3389/fpsyg.2020.582480.
- [4] A. Bhandari, A. Gupta, and D. Das, "Improvised apriori algorithm using frequent pattern tree for real time applications in data mining," *Procedia Computer Science*, vol. 46, pp. 644–651, 2015, doi: 10.1016/j.procs.2015.02.115.
- [5] X. Bai, J. Jia, Q. Wei, S. Huang, W. Du, and W. Gao, "Association rule mining algorithm based on spark for pesticide transaction data analyses," *International Journal of Agricultural and Biological Engineering*, vol. 12, no. 5, pp. 162–166, 2019, doi: 10.25165/j.ijabe.20191205.4881.
- [6] N. A. Azeez, T. J. Ayemobola, S. Misra, R. Maskeliūnas, and R. Damaševičius, "Network intrusion detection with a hashing based apriori algorithm using Hadoop MapReduce," *Computers*, vol. 8, no. 4, 2019, doi: 10.3390/computers8040086.
- [7] P. Gupta and V. Sawant, "A Parallel Apriori Algorithm and FP- Growth Based on SPARK," *International Conference on Automation, Computing and Communication 2021 (ICACC-2021)*, 2021, vol. 40, doi: 10.1051/itmconf/20214003046.
- [8] P. Gupta and D. V. Sawant, "A Map Reduce Based Parallel Algorithm," *Journal of University of Shanghai for Science and Technology*, vol. 22, no. 12, pp. 1375–1378, 2020, doi: 10.51201/jusst12486.
- [9] T. S. Yange, I. P. Gambo, R. Ikono, and H. A. Soriyan, "A Multi-Nodal Implementation of Apriori Algorithm for Big Data Analytics using MapReduce Framework," *International Journal of Applied Information Systems*, vol. 12, no. 31, 2020. [Online]. Available: <https://www.ijais.org/archives/volume12/number31/yange-2020-ijais-451868.pdf>
- [10] R. Zhang, W. Chen, T. C. Hsu, H. Yang, and Y. C. Chung, "ANG: a combination of Apriori and graph computing techniques for frequent itemsets mining," *The Journal of Supercomputing*, vol. 75, pp. 646–661, 2019, doi: 10.1007/s11227-017-2049-z.
- [11] J. M. Luna, P. F. -Viger, and S. Ventura, "Frequent itemset mining: A 25 years review," *WIREs Data Mining and Knowledge Discovery*, vol. 9, no. 6, 2019, doi: 10.1002/widm.1329.
- [12] S. Kumar and K. K. Mohbey, "A review on big data based parallel and distributed approaches of pattern mining," *Journal of King Saud University - Computer and Information Sciences*, vol. 34, no. 5, pp. 1639-1662, 2022, doi: 10.1016/j.jksuci.2019.09.006.
- [13] P. Singh, S. Singh, P. K. Mishra, and R. Garg, "A data structure perspective to the RDD-based Apriori algorithm on Spark," *International Journal of Information Technology*, vol. 14, pp. 1585-1594, 2022, doi: 10.1007/s41870-019-00337-3.
- [14] Y. Djenouri, D. Djenouri, A. Belhadi, and A. Cano, "Exploiting GPU and cluster parallelism in single scan frequent itemset mining," *Information Sciences*, vol. 496, pp. 363–377, 2019, doi: 10.1016/j.ins.2018.07.020.
- [15] K. -W. Chon, S. -H. Hwang, and M. -S. Kim, "GMiner: A fast GPU-based frequent itemset mining method for large-scale data," *Information Sciences*, vol. 439–440, pp. 19–38, 2018, doi: 10.1016/j.ins.2018.01.046.
- [16] H. Jordan, P. Subotić, D. Zhao, and B. Scholz, "Brie: A specialized trie for concurrent datalog," in *Proc. of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores*, pp. 31–40, 2019, doi: 10.1145/3303084.3309490.
- [17] N. K. Shinde and S. R. Gupta, "Architectures of Flynn's taxonomy-A Comparison of Methods," *IJISSET - International Journal of Innovative Science, Engineering and Technology*, vol. 2, no. 9, pp. 135–140, 2015. [Online]. Available: http://ijiset.com/vol2/v2s9/IJISSET_V2_I9_17.pdf
- [18] R. Ganesan, K. Govindarajan, and M. -Y. Wu, "Comparing SIMD and MIMD programming modes," *Journal of Parallel and Distributed Computing*, vol. 35, no. 1, pp. 91–96, 1996, doi: 10.1006/jpdc.1996.0071.
- [19] K. Sankaralingam, S. W. Keckler, W. R. Mark, and D. Burger, "Universal mechanisms for data-parallel architectures," *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, 2003, pp. 303-314, doi: 10.1109/MICRO.2003.1253204.
- [20] M. D. R. Klarqvist, W. Muła, and D. Lemire, "Efficient computation of positional population counts using SIMD instructions," *Concurrency and Computation: Practice and Experience*, vol. 33, no. 17, 2021, doi: 10.1002/cpe.6304.
- [21] J. Zhang, "Design of Multi-core Processor Software with Pipelining Strategy," *TELKOMNIKA Telecommunication Computing Electronics and Control*, vol. 12, no. 4, pp. 2950–2960, 2014. [Online]. Available: <https://ijeecs.iaescore.com/index.php/IJECS/article/view/3346/1507>
- [22] Anuradha T., S. Pasad R., and S. N. Tirumalarao, "Performance Evaluation of Apriori on Dual Core with Multiple Threads," *International Journal of Computer Applications*, vol. 50, no. 16, pp. 9–16, 2012, doi: 10.5120/7853-1088.
- [23] A. Hodijah and U. T. Setijohatmo, "Analysis of Frequent Itemset Generation Based on Trie Data Structure in Apriori Algorithm,"


- TELKOMNIKA Telecommunication, Computing, Electronics and Control*, vol. 19, no. 5, 2021, doi: 10.12928/telkomnika.v19i5.19273.
- [24] "Sets (Guava: google core libraries for java 19.0 API)," Accessed Dec. 17, 2021. [Online]. Available: [https://guava.dev/releases/19.0/api/docs/com/google/common/collect/Sets.html#powerSet\(java.util.Set\)](https://guava.dev/releases/19.0/api/docs/com/google/common/collect/Sets.html#powerSet(java.util.Set))
- [25] "Java - All possible combinations of an array - Stack Overflow," Accessed: Dec. 17, 2021. [Online]. Available: <https://stackoverflow.com/questions/5162254/all-possible-combinations-of-an-array>
- [26] W. Jiang, Q. Li, Z. Wang, and J. Luo, "A Framework of Concurrent Mechanism Based on Java Multithread," *Telecommunication Computing Electronics and Control*, vol. 11, no. 9, pp. 5395–5401, 2013. [Online]. Available: <https://ijeecs.iaescore.com/index.php/IJECS/article/view/2656/2876>
- [27] H. Guo and J. He, "A Fast Fractal Image Compression Algorithm Combined with Graphic Processor Unit," *TELKOMNIKA Telecommunication, Computing, Electronics and Control*, vol. 13, no. 3, pp. 1089-1096, 2015, doi: 10.12928/telkomnika.v13i3.1776.

BIOGRAPHIES OF AUTHORS






Ade Hodijah    received the Master degree from the Bandung Institute of Technology (ITB) in 2012. She is lecturer of informatics and computer engineering at State Polytechnic of Bandung. Her main research interests is related to knowledge discovery. She can be contacted at email: adehodijah@jtk.polban.ac.id.



Urip Teguh Setijohatmo    received bachelor degree in 1990 at University of Kentucky and the Master degree in 2000 at Universitas Indonesia. He is lecturer of informatics and computer engineering of State Polytechnic of Bandung. His main research interests is in data engineering. He can be contacted at email: urip@jtk.polban.ac.id.



Ghifari Munawar    received a master's degree in computer software engineering from the Bandung Institute of Technology (ITB) in 2012. He is a lecturer in informatics and computer engineering at the Bandung State Polytechnic (Polban), focusing on programming, software architecture, and implementation. The research topics related to software architecture, performance, web and mobile technology, and decision support systems. He can be contacted at email: ghifari.munawar@polban.ac.id.



Gita Suciana Ramadhanty    received the Diploma. degree in informatics and computer engineering from Politeknik Negeri Bandung, Bandung, Indonesia. She can be contacted at email: gita.suciana.tif415@polban.ac.id.