Task Graphs of Stream Mining Algorithms

Sayaka Akioka
Meiji University
4-21-1 Nakano, Nakano-ku
Tokyo, 164-8525, Japan
+81-3-5343-8305
akioka@meiji.ac.jp

ABSTRACT

Acceleration of huge data analysis, especially an analysis of huge, and fast streaming data is one of the major issues in recent computer science. Proper modeling, and understanding of streaming data analysis are indispensable for speed-up, scale out, and faster response time of streaming data analysis. Especially for the research on scheduling, or load balancing algorithms, a model of the target application truly impacts on the performance of the scheduling, or load balancing algorithms, however, there is no study on the realistic models, or the actual behaviors of streaming data analysis yet. This paper proposes a task graph for stream mining algorithms with some examples of actual applications. A task graph represents a workload of the target application with data dependencies, and control flows. This is the first proposal of task graphs for stream mining algorithms, and the task graphs play an important role as a benchmarking tool for the development of scheduling, or load balancing algorithms targeting on stream mining algorithms.

1. INTRODUCTION

Applications to process a massive amount of data, so-called "big data", is one of the recent hot research topics. Big data applications are sometimes considered to be quite similar with data intensive applications in high performance computing (HPC), however, the behaviors of applications in these two domains are quite different [9].

Big data applications utilize often stream mining algorithms, while data intensive applications process huge data in a batch. That is, big data application often tries to analyze data stream, which is a sequence of data arriving in chronological order, on the fly. As the data stream flows very fast, stream mining algorithms are developed with the purpose of the perfect analysis over such fast data flows. Once the delay of the analysis arises, and the analysis fails to keep up with the data arrival, the whole process will be forced to drop some of the arriving data. As many of the streaming analysis processes place emphasis on the real-time analysis in chronological order, a drop of the arrival data is highly critical.

As big data applications scale up with such a severe requirement for extremely low latency, big data applications become to run on the parallel and distributed computing environment such as the computing cloud. In order to exploit parallelism, and speed up the applications, scheduling is indispensable. Scheduling algorithms in parallel and distributed computing environment have been studied intensively for a long time especially in HPC, and these researches often validate, and compare the scheduling algorithms with task graphs. A task graph represents a workload of a target application, which is often synthetic workload generated randomly. As the quality of task graphs heavily impacts on validation of scheduling algorithms, the methodology to generate task graphs have been studied as well with a strong focus on data intensive applications in HPC.

This paper proposes task graphs generated from the actual implementations of stream mining algorithms in order to contribute to a development of effective, and practical scheduling algorithms for stream mining algorithms. The contributions of this paper are 1) the first proposal of task graphs for stream mining algorithms, 2) the practical and realistic workloads extracted from the existing implementations, 3) task graphs as representations of the behaviors of stream mining algorithms to open up unexplored problems for conventional scheduling algorithms, and 4) task graphs as a benchmarking tool to accelerate the development of scheduling algorithms for stream mining algorithms.

The rest of this paper is organized as follows. Section 2 gives a generic model of stream mining algorithms in order to clarify data dependencies of the process. Section 3 describes the procedure of task graph generation, and proposes a format of task graphs for stream mining algorithms. Section 4 overviews actual stream mining algorithms analyzed in this paper, and represents corresponding task graphs. Section 5 briefly introduces the related work, and Section 6 concludes this paper.

2. STREAM MINING ALGORITHMS

A stream mining algorithm is an algorithm specialized for a data analysis over data streams on the fly. There are many variations of stream mining algorithms, however, general stream mining algorithms share a fundamental structure, and a data access pattern as shown in Figure 1 [1].

A stream mining algorithm consists of two parts; a stream processing part, and a query processing part. First, the stream processing module picks the target data unit, which is a chunk of data arrived in a limited time frame, and executes a quick analysis over the data unit. The quick analysis here can be a preconditioning process such as a morphological analysis, or a word counting. Second, the stream processing module updates the data cached in one or more sketches with the latest results through the quick analysis. That is, the sketches keep the intermediate analysis, and the stream processing module updates the analysis incrementally as more data units are processed. Third, the analysis module reads the intermediate analysis from the sketches, and extracts the essence of the data in order to complete the quick analysis in the stream processing part. Finally, the query

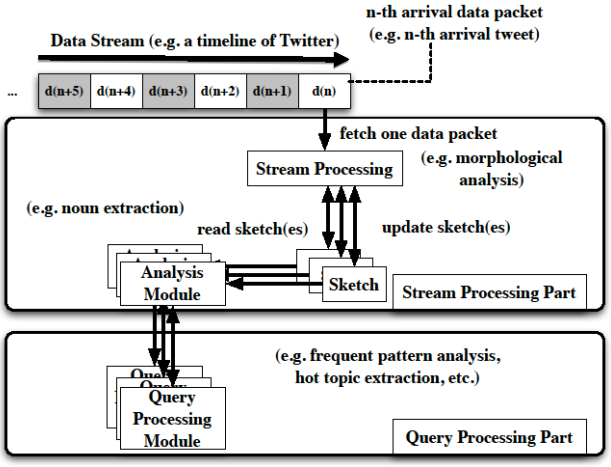


Figure 1. A model of stream mining algorithms.

processing part receives this essence for the further analysis, and the whole process for the target data unit is closed.

Based on the modeling above, we can conclude that the major responsibility of the stream processing part is to process each data unit for the further analysis, and that the stream processing part has the huge impact over the latency of the whole process. The stream processing part needs to finish the preconditioning of the current data unit before the next data unit arrives, otherwise, the next data unit will be lost as there is no storage for buffering the incoming data in a stream mining algorithm. On the other hand, the query processing part takes care of the detailed analysis such as a frequent pattern analysis, or a hot topic extraction based on the intermediate data passed by the stream processing part. The output by the query processing part is usually pushed into a database system, and there is no such an urgent demand for an instantaneous response. Therefore, only the stream processing part needs to run on a real-time basis, and the successful analysis over all the incoming data simply relies on the speed of the stream processing part.

The model of a stream mining algorithm shown in Figure 1 also indicates that the data access pattern of the stream mining algorithms is totally different from the data access pattern of so-called data intensive applications, which is intensively investigated in HPC. The data access pattern in the data intensive applications is a write-once-read-many [9]. That is, the application refers to the necessary data many times during the computation; therefore, the key for the speedup of the application is to place he necessary data close to the computational nodes for the faster data accesses. On the other hand, in a stream mining algorithm, a process refers to its data unit only once, which is a read-once-write-once style. Therefore, a scheduling algorithm for the data intensive applications is not simply applicable or the purpose of the speedup of a stream mining algorithm.

Figure 2 illustrates data dependencies between two processes analyzing data units in line, and data dependencies inside ne process. The left top flow represents the stream processing part of the preceding process, and the right bottom flow represents the stream processing part of the successive process. Each flow consists of the six stages; read from sketches, read from input, stream processing, update sketches, read from sketches, and analysis. An arrow represents a control flow, and a dashed arrow represents a data dependency.

In Figure 2, there are three data dependencies in total as follows, and all of these three dependencies are essential to keep the analysis results consistent, and correct.

1. The processing module in the preceding process should finish updating the sketches before the processing module in the successive process starts reading the sketches (Dep.1 in Figure 2).
2. The processing module should finish updating the sketches before the analysis module in the same process starts reading the sketches (Dep.2 in Figure 2).
3. The analysis module should finish reading the sketches before the processing module in the successive process starts updating the sketches (Dep.3 in Figure 2).

3. TASK GRAPH DEFINITIONS

As discussed in Section 2, a model of a stream mining algorithm has data dependencies both across the processes, and inside one process. Therefore, a task graph or a stream mining algorithm should consist of a data dependency graph, and a control flow graph. We already modeled both the data dependencies, and the control flows for stream mining algorithms in Section 2, however, a task graph is a finer grained model for a specific algorithm and implementation.

A data dependency graph is drawn via an analysis of the actual implementation of the target algorithm. Figure 3(a) is an example of a data dependency graph of the training stage of Naïve Bayes classifier[2] implemented by MOA project [8]. Figure 4 represents a pseudo code for the data dependency graph in Figure 3(a). A data dependency graph is a directed acyclic graph (DAG). In a data dependency graph, each node represents a basic block, or

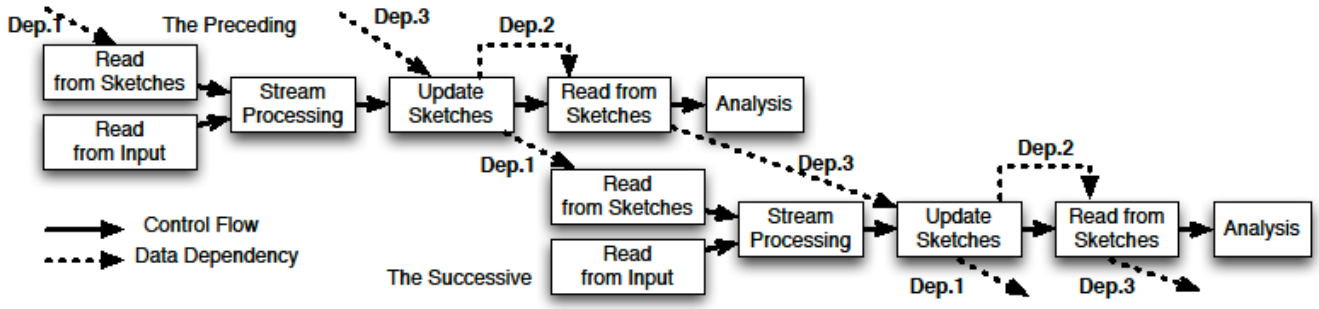
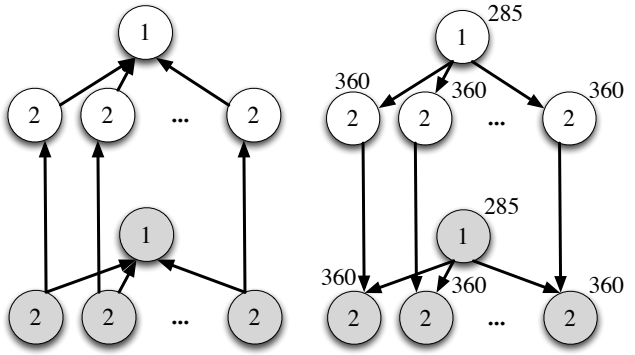


Figure 2. Data dependencies of the stream processing parts in two processes in line.



(a) **(b)**

Figure 3. The data dependency graph (a), and the control flow graph (b) for Naïve Bayes implementation of MoA.

for all training data **do**

(1) Fetch one training data *v*

for all attributes for *v* **do**

(2-1) Update the weight sum of this attribute.

(2-2) Update the mean value of this attribute.

end for

end for

Figure 4. The training stage of Naïve Bayes algorithm.

an equivalent chunk of codes in the actual implementation, and each array indicates a data dependency. If an arrow comes up from node A to node B, the arrow indicates that there is a data dependency between node A, and node B, and that the process represented by node B relies on the data generated by the process represented by node A for consistency of the analysis.

A data dependency graph in Figure 3(a) actually consists of two DAGs; a DAG with nodes in white, and a DAG with nodes in gray. Each DAG represents each process in Figure 2. That is, the DAG with white nodes in Figure 3(a) indicates the preceding process in Figure 2, and the DAG with gray nodes in Figure 3(b) indicates the successive process in Figure 2. The arrows between the two DAGs represent data dependencies between the two processes. In the case of stream mining applications, which is the most different point from conventional applications, the application continues running as long as a new data unit arrives. A DAG with nodes in a same color represents one process for one data unit, therefore, DAGs should lie in a line as many as the number of data the corresponding application processes. In this case, two DAGs are sufficient for the representation of the minimum unit of the repeated pattern in the application, and the data dependency graph does not contain any more redundant DAGs for simple but sufficient representation.

In a data dependency graph, each node has a number, and the number indicates that the particular node represents which basic block in the pseudo code, such as shown in Figure 4. In this example, node 1 represents the line starting with "(1)" in the pseudo code in Figure 4, and node 2 represents the lines starting

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:complexType name="NodeType">
  <xs:attribute name="id" type="xs:string" use="required" />
  <xs:attribute name="cost" type="xs:int" use="required" />
  <xs:attribute name="parallelism" type="xs:int" />
 </xs:complexType>

 <xs:complexType name="ArrowType">
  <xs:attribute name="id" type="xs:string" use="required" />
  <xs:attribute name="src" type="xs:string" use="required" />
  <xs:attribute name="dest" type="xs:string" use="required" />
 </xs:complexType>

 <xs:complexType name="DummyNodeType">
  <xs:attribute name="id" type="xs:string" use="required" />
  <xs:attribute name="cost" type="xs:int" fixed="-1" />
 </xs:complexType>

 <xs:complexType name="DDType">
  <xs:sequence maxOccurs="unbounded" minOccurs="2">
   <xs:element ref="arrow" />
  </xs:sequence>
 </xs:complexType>

 <xs:complexType name="CFType">
  <xs:sequence maxOccurs="unbounded" minOccurs="2">
   <xs:element ref="arrow" />
  </xs:sequence>
 </xs:complexType>

 <xs:complexType name="NodeListType">
  <xs:sequence>
   <xs:element ref="startNode" />
   <xs:sequence maxOccurs="unbounded" minOccurs="1">
        <xs:element ref="node" />
   </xs:sequence>
   <xs:element ref="endNode" />
  </xs:sequence>
 </xs:complexType>

 <xs:complexType name="TaskGraph">
  <xs:sequence>
   <xs:element ref="nodeList" />
   <xs:element ref="cf" />
   <xs:element ref="dd" />
  </xs:sequence>
 </xs:complexType>

 <xs:element name="node" type="NodeType" />
 <xs:element name="startNode" type="DummyNodeType" />
 <xs:element name="endNode" type="DummyNodeType" />
 <xs:element name="arrow" type="ArrowType" />
 <xs:element name="dd" type="DDType" />
 <xs:element name="cf" type="CFType" />
 <xs:element name="nodeList" type="NodeListType" />
 <xs:element name="taskgraph" type="TaskGraph" />
</xs:schema>
```

Figure 5. XML scheme for a task graph.

```
<?xml version="1.0" encoding="utf-8" ?>
<taskgraph>
 <nodeList>
  <startNode id="0" />
  <node id="1" cost="285" parallelism="1" />
  <node id="2" cost="360" parallelism="-1" />
  <endNode id="3" />
 </nodeList>

 <cf>
  <arrow id="cfa01" src="0" dest="1" />
  <arrow id="cfa12" src="1" dest="2" />
  <arrow id="cfa22" src="2" dest="2" />
  <arrow id="cfa23" src="2" dest="3" />
 </cf>

 <dd>
  <arrow id="dda10" src="1" dest="0" />
  <arrow id="dda21" src="2" dest="1" />
  <arrow id="dda22" src="2" dest="2" />
  <arrow id="dda32" src="3" dest="2" />
 </dd>
</taskgraph>
```

Figure 6. XML representation for the task graph in Figure 3.

with "(2-1)", and "(2-2)" in the pseudo code in Figure 4. Here, as determined from the pseudo code in Figure 4, the basic block indicated by node 2 is data parallel. Therefore, in the data dependency graph, several node 2s are located in the same level of the DAG. Logically, there is no limit of the number of node 2s in this case, therefore, an user can put node 2s as many as desired.

A control flow graph is also drawn via an analysis of the actual implementation of the target algorithm again, and the basic definitions are almost the same to the case of a data dependency graph. Figure 3(b) is a control flow graph for Naïve Bayes classifier, and the corresponding pseudo code is shown in Figure 4. Each node represents basic block again, however, each arrow in a control flow graph represents the order of the process of basic blocks. That is, in Figure 3(b), node 2 always has to be processed just after node 1 is completed. On the other hand, nodes without arrays in between do not have any ordering restriction. Therefore, these nodes can be executed in a shuffled order, or on the same stage. As the same to the data dependency graph, a control flow graph consists of the minimum but sufficient DAGs for the simplicity.

A control flow graph has a computational cost for each node. A computational cost shown in a control flow graph is the average of the actual computational costs measured in the actual computations, however, this version of the control flow graphs do not contain communication costs. As the control flow graphs here are fine-grained, it is not beneficial to scatter one control flow graph over the distributed computing environment. That is, pipelining control flow graphs according to the speed of the input data is a more realistic, and practical solution. Communication costs for pipelining in the distributed computing environment contains further discussions, and we reserve this topic for the future work.

Figure 5 is the XML schema for a task graph, and Figure 6 is an example representation of XML for Figure 3. Task graphs should

for all input data items **do**
  (1) fetch one input data v
  for all distinct items appeared **do**
    (2) create or update a border point for v
    (3) update summary
    (4) update frequency
    (5) delete obsolete border points
  end for
  (6) update pruning threshold
end for

Figure 7. A pseudo-code of top-k (min summary).

be represented also in XML according to this schema, and a designer of scheduling simulators can easily employ the task graph as a benchmark by reading this XML.

4. ACTUAL TASK GRAPHS

This section introduces task graphs extracted from the actual popular methodologies. One is top-k implemented as a Java 1.7 application. The other is Hoeffding tree algorithm[6], which is one of decision tree algorithms, and implemented as MOA module[8].

We implemented top-k based on min summary algorithm proposed by Lam et al.[7], and the base proposal by Calders et al. [3]. Figure 7 is the pseudo-code of the corresponding algorithm. Figure 8 (a) represents the extracted data dependency graph, and Figure 8(b) represents the extracted control flow graph.

As we already saw through the generic model of the stream mining algorithms in Section 2, each node processing one data unit basically depends on the results of the previous node. That is, each node is a consumer of the previous node. The exception is node 1 (data fetching), and node 6 (update of the pruning threshold). Especially, node 6 updates the pruning threshold based on the length of the summary, and node 6 has to wait for the elimination of the obsolete border points, which is node 5. On the other hand, the process represented from node 2 to node 5 is independent across the distinct items appeared during the observation, and this part is capable of parallel execution.

When we focus on the dependency between the preceding process, and the successive process, node 2 in the successive process depends on node 5 in the preceding process. Node 5 in the preceding process deletes the obsolete border points, while node 2 adds a new border point, or increment the counter of the existing border point according to the input. There is no dependency when node 2 adds a new border point, however, node 2 needs to decide which border point should be updated when node 2 increments the count of the existing border point. This is the reason why node 2 in the successive process behaves as a consumer of node 5 in the preceding process.

One more thing we would note here is that the computational cost of node 6 is relatively huge compared to the computational costs of the other nodes. The major reason of the heavy load of node 6 is that node 6 needs to calculate the maximum relative frequency of the least appeared item during the observation. Because of this process, node 6 is a consumer of node 5, needs to sweep all the data in the summary, and consumes more time for completion.

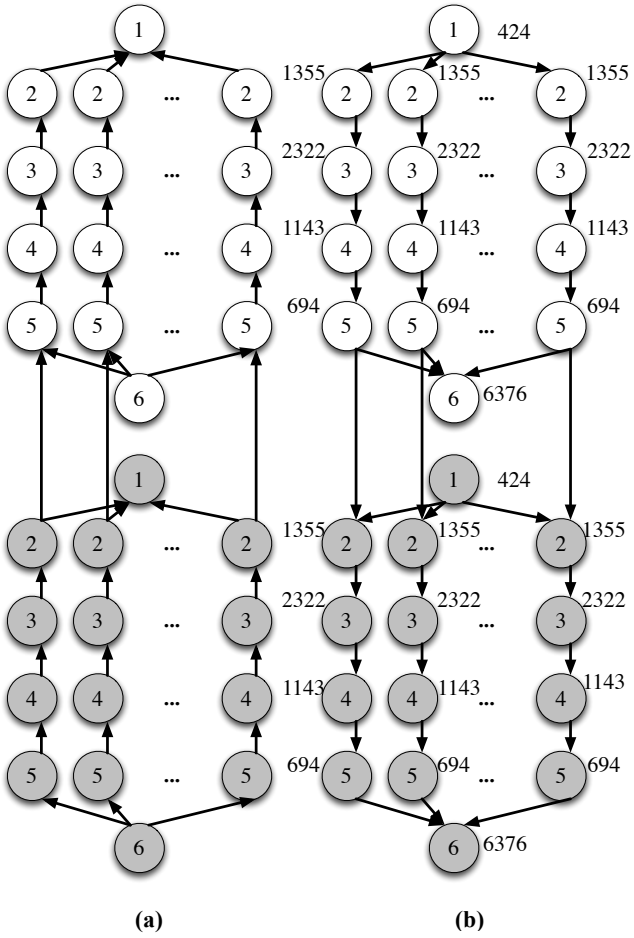


Figure 8. The data dependency graph (a), and the control flow graph for top-k (min summary).

This tendency of the computational cost implies that the execution in a pipeline is really effective for min summary algorithm. In fact, the computational cost of node 6 is almost equivarent to the total cost of the single path of nodes 1-5, and node 6 is independent from these nodes. Therefore, node 1-5, and node 6 are capable of running in a pipeline, and consumes almost the same computational time. That is, there is a chance to hide almost half of the execution time of the process time of one data unit, and improve the throughput by pipelining.

We also extracted a task graph of Hoeffding tree algorithm from MOA implementation. Figure 9 is the pseudo-code of the algorithm, and Figure 10 represents the extracted data dependency graph. The control graph is omitted for the page limitation. We skip the detailed discussion for the page limitation again, however, we can observe similar tendency of the application as we saw in the generic model in Section 2, Naïve Bayes in Section 3, and min summary algorithm in this section. One major difference from the previous cases is that node 1 depends on node 9, therefore, the effect of the pipelining is not huge compared to the other cases.

Here, we need to discuss computational costs in the control flow graph. This version of the task graph represents a computational cost as the average of actual executions. This is in a sort of the simplified model as the computational cost of stream mining algorithms easily varies depending on the input data. We need to develop the better methodology for the computational model, however, we reserve this issue as a future work.

```
Let HT be a tree with a single leaf (the root)
for all training data do
   (1)  Fetch one training data v, and sort v into leaf l
        using HT
   for all attributes for v do
      (2) Update sufficient statistics in l
   end for
   (3) Increment nl, the number of examples seen at l
   if n1 mod nmin = 0 and data seen at l not all of same
   class then
      (4) Compute Gl(Xi) for each attribute
      (5) Let Xa be attribute with highest Gl
      (6) Let Xb be attribute with second-highest Gl
      (7) Compute Hoeffding bound
      if Xa != Xb and (Gl(Xa) − Gl(Xb) > ε or ε < τ)
      then
         (8) Replace l with an internal node that splits on Xa
         for all branches of the split do
            (9) Add a new leaf with initialized sufficient
                statistics
         end for
      end if
   end if
end for
```

Figure 9. A pseudo-code of a training tree of Hoeffding tree algorithm.

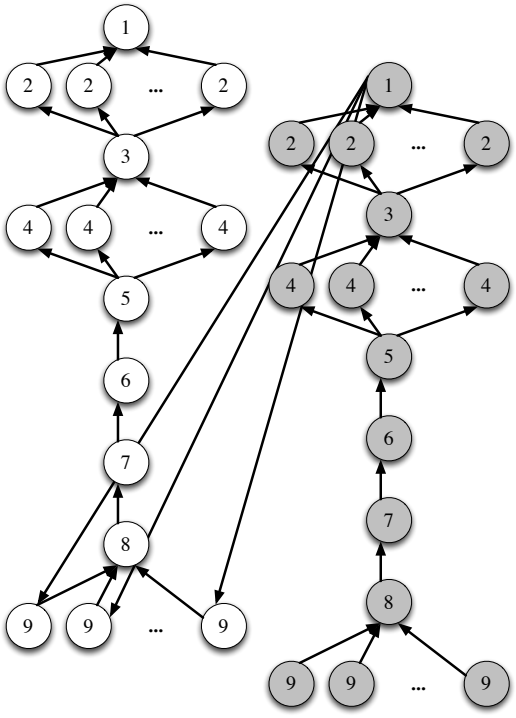


Figure 10. A data dependency graph for Hoeffding tree algorithm.

5. RELATED WORK

There are several studies on task graph generation, mainly focusing on random task generation. A few projects reported task graphs generated based on the actual well-known applications, however, those applications are from numerical applications such as Fast Fourier Transformation, or applications familiar to HPC community for a long time.

Task Graphs for Free (TGFF) provides pseudo-random task graphs [5,11]. TGFF allows users to control several parameters, however, generates only directed acyclic graphs (DAGs) with one or multiple start nodes, and one or multiple sink nodes. Each task graph is assigned a period, and a deadline based on the length of the maximum path in the graph, and the user specified parameter.

GGen is another random task graph generator proposed by Cordeiro et al [4]. GGen generates random task graphs according to the well-known random task generation algorithms. In addition to the graph generator, GGen provides a graph analyzer, which characterizes randomly generated task graphs based on the longest path, the distribution of the out-degree, and the number of edges.

Task graph generator provides both random task graphs, such as Fast Fourier Transformation, Gaussian Elimination, and LU Decomposition [12]. The random task graph generator supports variety of network topologies, including star, and ring. Task graph generator also provides scheduling algorithms as well.

Tobita et al. proposed Standard Task Graph Set (STG), evaluated several scheduling algorithms, and published the optimal schedules for STG [10,13]. STG is a set of random task graphs, which are ready to download. Tobita et al. also provides task graphs from numerical applications such as a robot control programs, a sparse matrix solver, and SPEC fpppp.

Besides the studies on task graph generation, Cordeiro et al. pointed out that randomly generated task graphs can create biased results, and that the biased results can mislead the analysis of scheduling algorithms[4]. According to the experiments by Cordeiro et al., a same scheduling algorithm man obtain a speedup of 3.5 times only by changing the graph generation algorithm for the performance evaluation.

Random task graphs contributes positively for evaluation of scheduling algorithms, however, do not perfectly cover all the domains of parallel and distributed applications as Cordeiro et al. figured out in their work. Especially for stream mining applications, which focus on in this paper, the characteristic of the application behaviors are quite different from the characteristic of the applications familiar to the conventional HPC community as we discussed in Section 2. Task graphs generated from the actual stream mining applications have profound significance in the better optimization of the applications in parallel computing environment for wider area of applications.

6. CONCLUSION

This paper proposed task graphs for stream mining algorithms. This is the first clear proposal of task graphs modeling stream mining algorithms, and the task graphs are extracted from the actual implementations of the popular existing methodologies. Task graphs proposed in this paper play an important role as the benchmarking tool to evaluate scheduling algorithms, or load balancing algorithm, which is indispensable for the research of scheduling, or load balancing algorithms truly effective for stream mining algorithms. In fact, in this paper, the proposed task graphs represent apparently different characteristics, and dependencies compared to the data intensive applications in HPC, and this fact points out we need to consider scheduling methodologies focusing on stream mining algorithms. For the better set of task graphs, we are working on more stream mining algorithms.

7. REFERENCES

[1] Akioka, S., Muraoka, Y., Yamana, H., Data access pattern analysis on stream mining algorithms for cloud computation. In *Proceedings of the 2011 International Conference on Parallel and Distributed Processing (PDPTA2011)* (Las Vegas, USA, July 18-21, 2011), 2011, 36-42.

[2] Bifet, A., Holmes, G., Pfahringer, B., Karen, P., Kremer, H., Jansen, T., Seidl, T., MOA: Massive online analysis, a framework for stream classification and clustering. *Journal of Machine Learning Research (JMLR)*, Workshop and Conference Proceedings Vol. 11: Workshop on Application of Pattern Analysis, 2010.

[3] Calders, T., Dexters, N., Goethals, B., Mining Frequent Itemsets in a Stream. In *Proceedings of 2007 IEEE International Conference on Data Mining* (ICDM2007) (Omaha, USA, October 28-31, 2007), 2007.

[4] Corderiro, D., Mounie, G., Perarnau S., Trystram, D., Vincent, J. M., Wagner, F., Random graph generation for scheduling simulations. In *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques (SIMUTools'10)* (Torremolinos, Spain, March 15-19, 2010), 2010.

[5] Dick, R. P., Rhodes D. L., Wolf, W., TGFF: Task graphs for free. In *Proceedings of International Workshop on Hardware/Software Codesign* (Seattle, USA, March 15-18, 1998), 1998, 97-101.

[6] Domingos, P., Hulten, G., Mining High-Speed Data Streams. In Proceedings of *The 6th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (KDD'00) (Boston, USA, August 20-23, 2000), 2000, 71-80.

[7] Lam, H. G., Calders, T., Mining top-k frequent items in a data stream with flexible sliding window. In *Proceedings of The 16th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (KDD'10) (Washington DC, USA, July 25-28, 2010), 2010.

[8] McCallum A., Nigram, K., A comparison of event models for Naïve Bayes text classification. In *Proceedings of AAAI-98 Workshop on 'Learning for Text Categorization'* (Madison, USA, July 26-27, 1998), 1998.

[9] Raicu, I., Foster, I. T., Zhao, Y., Little, P., Moretti, C. M., Chaudhary, A., Thain, D. The quest for scalable support of data intensive workloads in distributed systems. In *Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing (HPDC2009)* (Munich, Germany, June 11-13, 2009), 2009, 207-216.

[10] STG, Standard task graph set. http://www.kasahara.elec.waseda.ac.jp/schedule/index.html.

[11] TGFF. http://ziyang.eecs.umich.edu/~dickrp/tgff/.

[12] TGG, Task graph generator. http://taskgraphgen.sourceforge.net/.

[13] Tobita T., Kasahara, H., A standard task graph set for fair evaluation of multiprocessor scheduling algorithms. *Journal of Scheduling*, Volume 5, Issue 5, 2002, 379-394.