RC25181 (D1107-003) July 5, 2011
Computer Science

IBM Research Report

Application-Agnostic Generation of Synthetic Task Graphs for Stream Computing Applications

Deepak Ajwani, Shoukat Ali, John P. Morrison
IBM Research
Smarter Cities Technology Centre
Mulhuddart
Dublin 15, Ireland

Research Division
Almaden - Austin - Beijing - Cambridge - Dublin - Haifa - India - T. J. Watson - Tokyo - Zurich

Application-Agnostic Generation of Synthetic Task Graphs for Stream Computing Applications

Deepak Ajwani Shoukat Ali John P. Morrison

Abstract

In order to process massive amounts of streaming data in real time, high-performance computing systems and software platforms are being developed. However, the pool of available streaming applications is small and really large streaming applications are even more scarce. As such, there is a need for synthetic generators of task graphs for performance evaluation and comparative analysis of various algorithmic techniques and hardware parameters.

In this paper, we identify the key properties of stream-computing graphs that are shared by most (if not all) streaming graphs, irrespective of their specific application domain. We then propose a new approach to generate task graphs and show that the generated graphs satisfy the identified properties.

1 Introduction

Many applications such as real-time analysis of financial and medical data [6], audio/video applications [8, 9], continuous database queries [1] and intelligent transportation systems [3], consists of processing continuous, high-volume data-streams in real time. To satisfy the requirements of these applications, high-performance stream computing systems are being developed. The design of such systems involve choosing between various algorithmic techniques for creating the software platform as well as choosing the right set of hardware parameters. Such choices are based on performance measurements collected from a large number of simulations. Ideally, these simulations should be based on the real application data. However, since stream computing is still a nascent technology, the pool of mature applications is small. The number of really large applications (for evaluation of stream computing applications at scale) is even smaller. As such, there is a need for generating large scale synthetic data-sets that emulate the properties of the real streaming data. This paper focusses on generating synthetic computational task graphs for stream computing applications.

Streaming Graphs In the computational task graphs of streaming applications (hereafter referred as streaming graphs), vertices denote the various operations being performed on the data streams (such as sampling,

filtering, copying etc.) and the edges represent the data streams. The weight of a vertex is proportional to the computational load associated with it and the weight of an edge denotes the relative rate of the corresponding stream. More formally, we select the average streaming rate of an arbitrary input stream (from an external source) as a base rate. For an edge e of the streaming graph, its weight $w(e)$ is the ratio between the average rate of the stream corresponding to e and the base rate. Similarly, we assign the average number of operations required to process one element in the steady state as a base and let the weight of a vertex v denote the ratio between the average number of operations required for the kernel corresponding to v and the base.

A major challenge in designing application-agnostic generators for streaming graphs is to determine the key properties that a typical instance of a streaming graph should satisfy and to gather the relevant statistics to fill up the specifications. There is little work so far on understanding the properties of streaming graphs. In order to generate synthetic graphs that emulate the properties of real streaming graphs, we first need to characterize these properties. In Section 3, we record the key properties of streaming graphs that form the basis of our generator. Our simple framework for generating synthetic streaming graphs is described in Section 4. In Section 5, we show that the generated graphs indeed satisfy the properties observed in Section 3.

2 Related Work

There is a large body of work related to generation of random graphs satisfying various constraints. While some of this can be useful for generating graphs with one or more properties of real streaming graphs, we are interested in generating random graphs that emulate most observed properties. The most relevant work in this direction is the work of Ajwani et al. [2]. However, their generation framework relies on application-specific input from the user such as degree distribution, the mix of kernel types, and number and length of cycles and then uses sophisticated techniques to meet these additional constraints. In contrast, our graph generators are application-agnostic, i.e., the graph generation framework does not rely on any domain-specific information from the user. Also, our approach is simpler and by not focusing on matching the exact numbers of kernel types and degree distribution, it matches the main properties of streaming graphs much better.

Regarding the characterization of streaming graph properties, a closely related work is that of characterizing stream programs from the StreamIT benchmark [11] in the context of language and compiler design. Some characteristics observed for these sub-routines can easily be generalized to whole

applications such as the relative proportion of kernel mixes with different weights. But in general, the global properties of these programs need not hold for entire streaming applications. Also, the StreamIT framework restricts the streaming graphs to be series-parallel and this limitation "makes code often unnatural and sometimes infeasible" [10]. Streaming graphs from real applications do not necessarily satisfy this constraint. Therefore, we do not restrict our framework to generate only series-parallel graphs, although we do ensure that our graphs have a large series-parallel subgraph.

3 Key Characteristics of Streaming Graphs

In this section, we record the key properties of the streaming graphs. Based on various discussions with researchers who have first-hand experience with streaming applications, we conjecture that most streaming graphs will satisfy these properties. However, at the moment, the pool of available streaming applications is too small to obtain statistically relevant measurements supporting these properties. A subset of these properties have also been noted in a recent paper [2]. The statistics on kernel mixes are based on the characterization of StreamIT benchmark [11, 10].

1. Streaming graphs are very sparse. Since each edge represents a high-volume, continuous data-flow, a large number of edges imply communication of massive amount of data. This is quite likely a result of a poor design choice.

2. There are no vertices with more than 1 in-degree and more than 1 out-degree. All vertices fall into one of the three types:

    - Filters: Vertices with in-degree 1 and out-degree 1. In general, filters can do any kind of data transformation including but not restricted to sampling, filtering, sliding window computations. A special case of identity filters merely pass the data as they receive it.
    - Split: Vertices with in-degree 1 and out-degree greater than 1. The splits are subdivided into following categories: Copy splits that copy the input stream to output streams; Round-robin or If-else distributors that distribute the input stream into output streams.
    - Join: Vertices with in-degree greater than 1 and out-degree 1

3. A large majority of the vertices are filters. Around 35% of splits are copy splits – they copy the input stream to output streams. The remaining splits are mostly distributing splits – they distribute the input stream (in a round-robin way, based on value of elements etc.)

into the output streams. Most joins merge the input streams in some way, i.e., their output stream rate is the sum of input stream rates.

4. Streaming graphs are mostly acyclic. Note that there may be cycles involving control signals that are short data-flows, but we do not model them as edges in our computational task graph. The edges in our definition of streaming graphs strictly correspond to continuous, high-volume data streams.

5. For any vertex pair (x, y), all paths from x to y have roughly the same length, where the length is defined as the number of edges in the path independent of the weights on the constituent edges.

6. In a typical stream computing application, computationally intensive tasks are usually performed towards the end of the computation process after the initial kernels have sampled and reduced the data volume significantly. In other words, the weights on nodes closer to the sink are significantly higher than those that are closer to the source. On the other hand, the weight of edges that reflect the data-flow volume, decreases as we traverse from sources to sink in the directed acyclic graphs. In particular, the filter vertices in the early part of the computation process significantly reduce the data rate.

7. There are more splits than joins close to the sources and there are more joins than splits closer to the sinks.

4 Synthetic Graph Generation Framework

In this section, we describe the general framework for generating the streaming graphs emulating the above properties. As our generation framework is application-agnostic, the only user parameter it takes is n – the number of vertices in the generated graph. The output is a graph in either the dot format [5] or in the format of 10th DIMACS implementation challenge [4], which is also the format for partitioning libraries such as METIS [7]. It can generate both the directed and the corresponding undirected versions of the graph.

Our framework first generates the "core" of the streaming graph with $\Theta(n^{\frac{1}{3}})$ vertices. To create this core, we first generate an acyclic series-parallel multi-graph with $\Theta(n^{\frac{2}{3}})$ edges and then add $\Theta(n^{\frac{1}{3}})$ random edges, preserving the acyclicity of the multi-graph. This ensures that the final graph has a big series-parallel subset, though it is not series-parallel itself.

In order to generate a series-parallel multigraph, we start with two vertices and $O(n^{\frac{1}{3}})$ edges from one to another. We then repeatedly add vertex pairs till we have $\Theta(n^{\frac{1}{3}})$ vertices. Adding a vertex pair (x, y) is done by selecting an existing edge (u, v) uniformly at random, removing (u, v), adding

$(u, x), (y, v)$ a single time and the edge (x, y) with a multiplicity l. The number l is chosen between 1 and $2n^{\frac{1}{3}}$, uniformly at random.

Next, we decompose the vertices with multiple in and multiple out-edges into a split-join pair. For each vertex v with indegree and outdegree greater than 1, we decompose it into a split vertex x and a join vertex y such that all in-edges to v become in-edges to x, all out-edges from v become out-edges from y and there is an edge from x to y. In the resultant graph, let $L(v)$ be the longest path from a source vertex (with in-degree 0) to $v \in V'$ and let $D(e) = L(v) - L(u)$ for an edge $e = (u, v)$. We then replace each edge $e = (u, v)$ by a path $P(e)$ from u to v of length proportional to $D(e)$ consisting of newly created filter vertices. This ensures that in the resultant graph, all paths between two vertices have roughly the same length. As we show in Section 5, this also ensures that a large majority of vertices in the graph are filter vertices. We then sort the edges in the resultant graph and remove duplicates.

We then assign weights to vertices and edges. Let $L'(v)$ be the longest path to v in the generated topology. We assign the weight of each vertex v to be $L'^2(v) + 1$. Next, we divide the splits and joins into sub-categories and assign weights to edges. Based on the statistics observed in [11] for StreamIT benchmark sub-routines, we mark 35% of splits as copying splits and the remaining as distributor splits. For the copying splits, the weight of output edges is the same as the weight of the input edge. For the distributor splits, the input weight is equally divided among the output edges. All joins add the weights of in-edges to the out-edge. The weight of the out-edge of a filter v is half that of its in-edge with probability $1/(\alpha \cdot L'(v) + 1)$ (for a small constant $\alpha < 1$) and equal to its in-edge with the remaining probability. For the source vertices with a single out-edge, we assign it a weight of 1. For a source vertex v with multiple out-edges, we declare the weight of out-edges to be 1 with probability 0.35 and $1/out_degree(v)$ with probability 0.65. As we show in Section 5, this edge assignment results in edge weights that decrease with the length of longest paths to its tail.

5 Experiments

In this section, we report our experiments for generating graphs with 10,000 vertices, averaged over 1000 runs. The generated graphs have an average of 9999.59 vertices, thereby getting very close to the user specification. The average number of edges is 10262.3, which implies that the generated graphs are extremely sparse (with an average out-degree of 1.026) satisfying property 1. Since we decompose the multi-input and multi-output vertices into split-join pairs and introduce only filter vertices afterwards, we do not have any vertex with more than 1 in-degree and 1 out-degree in our graph, satisfying property 2. Out of 9999.59 vertices, 9937.59 are filters, 29.73 are joins

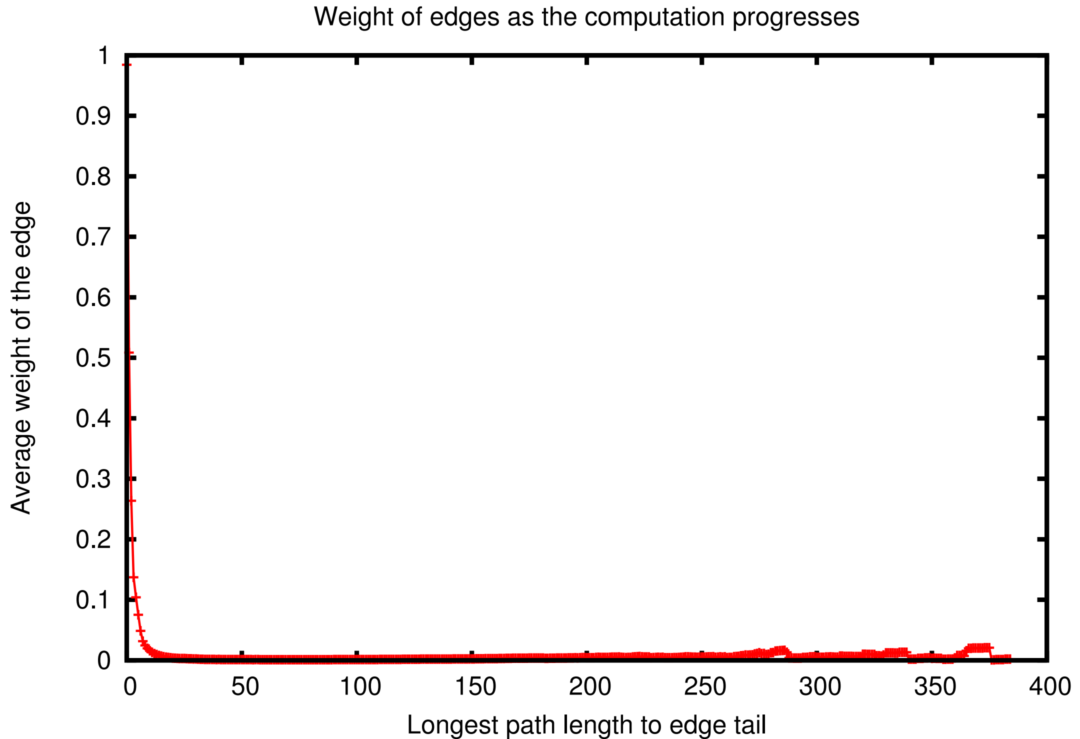

Figure 1: Weight of edges as computation progresses

and 30.27 are splits. There is only 1 source vertex in all the graphs. Out of the 30.27 splits, 10.65 (35.18%) are copy splits and 19.61 are distribute splits. All joins are merging joins. This satisfies the statistics observed in property 3. The generation process ensures that the generated graph is acyclic (property 4).

In order to verify that the property 5 is satisfied by our generated graphs, we measure the difference between longest and shortest path from a source to an internal vertex. Consider a vertex pair (u, v) with a big path difference. The longest path from a source to v has to be lengthier than the path from source to u followed by the longest path from u to v, while the shortest path has to be shorter than the path from source to u and shortest path from u to v. Thus, the difference between longest and shortest path to v will be greater than the maximum difference between the lengths of various paths from u to v. We observe that over all vertices, the average difference between longest and shortest path is only 1.65, compared to the average longest path length of 72.78. This implies that the average path difference between vertex pairs is smaller than 1.65.

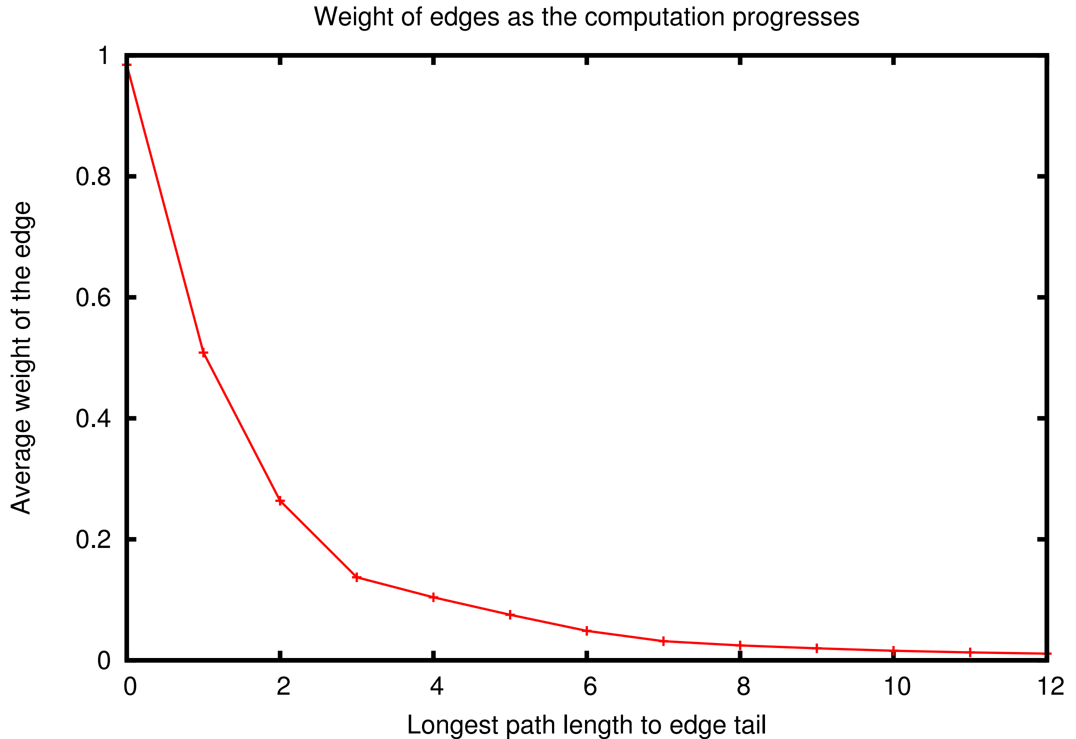

Figure 2: Weight of edges as computation progresses: Zoomed Initial part

To verify property 6, we first define the tail distance of an edge (u, v) to be the length of the longest path from a source in the DAG to the tail u. Intuitively, the tail distance measures where the edge belongs in the computation process. A smaller tail distance means that the edge is very close to the source and a larger tail distance means that the data-flow corresponding to the edge happens at a later stage in the computation process. Figure 1 shows how the weight of edges varies with the tail distance. The weight of edge represents the total weight of all the edges with the same tail-distance in all the 1000 runs of this experiment, divided by the number of such edges. As depicted in Figure 2 (which shows the early part of Figure 1 in greater detail), the averaged weight drops significantly very early in the computation process and then stays at around 1% of the early weight, even as the joins in the later part aggregate the weights of their in-edges. Thus, our generated graphs also satisfy property 6. Note that the shape of the curve can be partly controlled by the parameter α that determines the probability of a filter vertex to reduce the data-rate. The curve in Figure 1 is obtained using $\alpha = 0.25$.

As for property 7, we sorted the vertices by their shortest path from the source and considered the first 5% vertices. We found that 1.2% of these vertices were splits and only 0.2% were joins as compared to 0.3% splits and 0.29% joins overall. Similarly, we sorted the vertices by their shortest path from the sinks and considered the first 5% vertices. We found 0.6% of these vertices to be joins and 0.08% to be splits, thereby showing that in the generated graphs, there are significantly more splits than joins closer to sources and more joins than splits closer to sinks.

6 Conclusion

We have proposed a simple framework for generating application-agnostic streaming graphs. Our graphs emulate the properties expected of real streaming graphs quite well. We expect our framework to fill the need for synthetic streaming graphs for making software and hardware design choices related to the design of stream-computing systems. In particular, we intend to use our framework for designing a graph partitioning software for stream computing systems.

Acknowledgements

The authors would like to thank Dilma M Da silva, Qi Liu, Yoonseo Choi, Abhirup Chakraborty and Rolf Riesen for many helpful discussions and valuable feedback on an earlier version of the draft. The research of the first author is partially supported by an EPS grant from IRCSET and IBM.

References

[1] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.

[2] Deepak Ajwani, Shoukat Ali, Kostas Katrinis, Cheng-Hong Li, Alfred J. Park, John P. Morrison, and Eugen Schenfeld. A flexible workload generator for simulating stream computing systems. In *MASCOTS'11: Proceedings of the ninteenth annual IEEE Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 2011.

[3] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S. Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear road: a stream data management benchmark. In

VLDB '04: Proceedings of the Thirtieth International Conference on Very Large Data Bases, pages 480–491. VLDB Endowment, 2004.

[4] 10th DIMACS implementation challenge - Graph partitioning and graph clustering. `http://www.cc.gatech.edu/dimacs10/`.

[5] Dot tutorial and specification. `http://www.graphviz.org/Documentation.php`.

[6] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, and Myungcheol Doo. Spade: the System S declarative stream processing engine. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 1123–1134, New York, NY, USA, 2008. ACM.

[7] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1999.

[8] B. Khailany, W.J. Dally, U.J. Kapasi, P. Mattson, J. Namkoong, J.D. Owens, B. Towles, A. Chang, and S. Rixner. Load distributing for locally distributed systems. *IEEE Micro*, 21(2):35–46, March 2001.

[9] S. Rixner. *Stream processor architecture*. Kluwer Academic Publishers, 2002.

[10] William Thies. *Language and Compiler Support for Stream Programs*. PhD thesis, Massachusetts Institute of Technology, 2009.

[11] William Thies and Saman P. Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. In *19th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 365–376, 2010.