Dynamic Shortest Paths using JavaScript on GPUs

Anurag Ingole*
IIT Madras, India
anurag@cse.iitm.ac.in

Rupesh Nasre
IIT Madras, India
rupesh@cse.iitm.ac.in

***Abstract*—Information on the internet is growing rapidly and its processing needs high-speed infrastructure, both in hardware and software. JavaScript is now an integral ingredient of web applications which perform tasks ranging from error checking in online forms to processing Google maps. Due to their interactive nature, performance of JavaScript applications is critical, especially while handling huge volumes of evolving data. Therefore, parallelization of JavaScript code has been pursued in the recent past. In this work, we target GPU parallelization of dynamic graph algorithms on GPUs. We present implementation and achieve effective parallelization of dynamic single source shortest path computation. We compare the incremental, decremental and fully dynamic versions against their static counterpart and show that upto about 10% of updates, dynamic processing on GPUs is beneficial.**

***Index Terms*—GPU, SSSP, JavaScript, WebCL, dynamic, Node.js**

I. Introduction

JavaScript is a dynamic programming language. It is most commonly used as part of web browsers, whose implementations allow client-side scripts to interact with the user, control the browser, communicate asynchronously, and alter the document content that is displayed [1]. It is also used in server-side network programming with runtime environments such as `Node.js` [2], game development and the creation of desktop and mobile applications. With the rise of the single-page web applications and JavaScript-heavy sites, it is increasingly being used as a compile target for source-to-source compilers from both dynamic as well as static languages. JavaScript is predominantly sequential, and web applications until recently have been unable to utilize hardware parallelism. Considering the parallelism support by current hardware; web experience can be evolved to the next level if JavaScript is made to run in parallel.

JavaScript is used in several web-products that operate on an underlying graph, such as Google Maps and Facebook. Google Maps processes a network of junctions and cities to find the shortest paths from one point to another via a given mode of travel. Facebook creates a social network of friend connections, and allows retrieving attributes of various graph vertices (such as the update stream of a friend). In real-life, these graphs are dynamic in nature, that is, new vertices and edges keep getting added or removed from the underlying graph. We target such dynamic graphs in our work. As a preliminary study, we investigate the effect of computing dynamic shortest paths in JavaScript in parallel on a GPU for large real-world graphs. Thus, instead of recomputing the shortest paths for the modified graph, the goal is to perform only a small amount of parallel processing to get the modified shortest paths.

In this work, we implement a work-efficient fully dynamic, that is, incremental and decremental single source shortest path (SSSP) algorithm in JavaScript. The incremental algorithm follows the same methodology as that of the static algorithm, while the decremental processing involves extra care to be taken. This paper presents the ideas implemented on SSSP and BFS, but these ideas are general enough to be extended to other graph analytics algorithms like graph coloring, finding connected components, computing Page Rank etc.

II. Related Work

A few implementations for parallelizing JavaScript on multi-core systems and GPUs exist, such as WebCL, ParallelJS and RiverTrail.

A. JavaScript Parallelization

Parallel.js [3] is a tiny library for multi-core processing in JavaScript. It was created to take full advantage of the ever-maturing web-workers API. It uses the support of web workers provided by native browser to run script in parallel. Web workers help in writing multi threaded JavaScript code. Hence different bits of JavaScript code may be running at a particular instance of time. The level of parallelism achieved by this implementation is limited since scheduling of JavaScript thread on multi core

processor is dependent on OS and there is a limitation on the number of threads for multi core processors.

ParallelJS [4] is used for flexible mapping of JavaScript onto heterogeneous systems that have both CPUs and GPUs. The framework includes a front-end compiler, construct library and a runtime system. JavaScript programs written with high-level constructs are compiled to GPU binary code and scheduled to GPUs by the runtime. The program can be executed on either the CPU using the native JavaScript compiler or translated to PTX and executed on the GPU.

RiverTrail [5] is a JavaScript library and a Firefox add-on that together provide support for data-parallel programming in JavaScript, targeting multi-core CPUs and GPUs via OpenCL. The central component of RiverTrail is the ParallelArray type which models ordered collections of scalar values. ParallelArray objects support primitives such as map, reduce, scan, and combine, which are amenable to parallelism.

The WebCL [6] project exposes OpenCL into JavaScript, allowing parallel computation on modern GPUs, multi-core CPUs and many core accelerators. WebCL supports all the functionality provided by OpenCL. As OpenCL targets a wide variety of parallel architectures compared to Nvidia CUDA, we base our implementation on WebCL.

B. Parallel SSSP

There are many implementations of parallel static graph algorithms on a variety of architectures, including distributed-memory supercomputers [7], shared-memory supercomputers [8], and multicore machines [9]. Harish and Narayanan [10] describe CUDA implementations of graph algorithms such as BFS and single-source shortest paths computation.

There exists a relatively large body of work on speeding up processing of evolving graphs [11], [12], [13], [14], [15]. While Chronos [11] introduces a novel memory layout for evolving graphs to improve cache locality during serial or parallel graph processing, much of the other work restricts type of queries or are designed for a specific algorithm. For instance, Ren et al. [12] and Kan et al. [13] consider queries that depend upon the graph structure alone while Desikan and Srivastava [14] exploit specific properties of the Page Rank algorithm. None of these works amortize processing costs as we do.

III. Dynamic SSSP Computation

We work with directed weighted graphs with positive edge weights. We first explain how dynamic graphs are stored in memory, and then discuss incremental, decremental and fully dynamic SSSP.

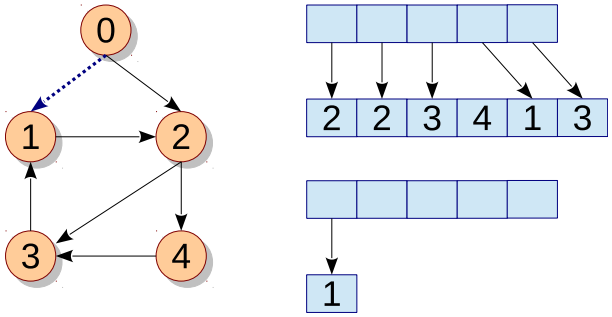

Figure 1. Graph with dynamic CSR edge representation

A. Dynamic Graphs and their Representation

Dynamic graphs undergo series of modifications like insertion and deletion of edges and vertices, as well as modifications to vertex and edge attributes. Since insertion of a vertex may be simulated by adding an edge to a disconnected vertex, we consider insertions and deletion of edges alone. This way, we can simulate edge weight modification as a combination of edge deletion and edge insertion. Thus, we do not reduce the generality of application.

The graphs are represented in compressed sparse row (CSR) format where entries in the edge array are pointed to by the vertices in the vertex array. Additional weight array of the same size as the edge array is also maintained to store weights of the corresponding edges. The set of newly inserted edges is maintained in a new CSR array. For each edge to be deleted from the graph, its weight in the weight-array is increased to `MAX`. An example graph, its CSR representation and the dynamic CSR representation are shown in Figure 1.

The worklist based parallel SSSP algorithm for static graphs discovers new minimum paths by propagating shortest paths through the graph. A vertex is added to the worklist if and only if its distance has reduced in the current iteration. Continuing this processing until the worklist becomes empty ensures shortest distance computed for each vertex in the graph. The information computed by the static version is used as the base distances by the dynamic version.

B. Incremental SSSP

In the incremental processing updates distances on new edge addition. A useful property of incremental SSSP is that the information is always propagated in the forward direction (away from the updated edge). Thus,

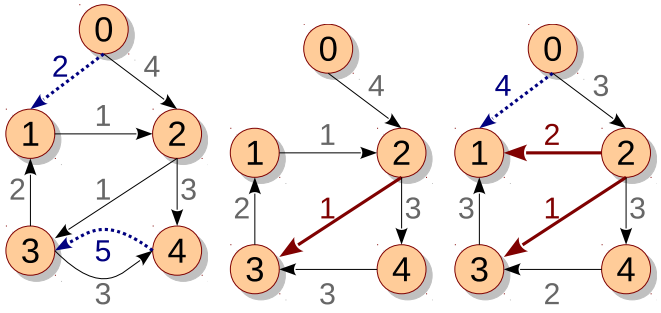

Figure 2. (a) Incremental update (b) Decremental update (c) Incremental + decremental update

whenever a new edge $u \rightarrow v$ is added, the distance of the vertex u never changes, the distance of v may change, and the distance of any vertex that is unreachable from v would not change. Node v is added to the worklist if its current distance (computed from the previous static computation) is larger than the sum of u's distance and the weight on new edge $u \rightarrow v$. When v is processed, its neighbors may get added to the worklist, and so on. All the nodes in the worklist are processed in the next iteration to propagate the newly found shortest paths. In the OpenCL kernel for incremental SSSP all the worklist vertices are processed in parallel. The kernel is launched repeatedly until no new shortest path can be discovered, i.e., until the worklist becomes empty.

Example. Consider the graph in Figure 2(a) where two new edges are added: $0 \rightarrow 1$ and $4 \rightarrow 3$. The processing starts with both 0 and 4 in the worklist. The newly added edge $0 \rightarrow 1$ with weight 2 reduces the current distance of vertex 1, and 1 is added to the worklist. The distance is now propagated to all the children of 1 which causes change in the distance of 2 (from 4 to 3). In the next iteration, the distances of neighbors 3 and 4 are updated (from 5 to 4 and from 7 to 6 respectively). At this step, no more nodes get added to the worklist, and the processing stops. The example also shows that depending upon where in the graph the new edge is added, the amount of processing may differ.

C. Decremental SSSP

Decremental processing for SSSP has a non-triviality that deletion of the shortest-path edge requires finding the next shortest path. Further, this needs to be done potentially for all the vertices reachable from the deleted edge. This processing becomes complicated because the next shortest path may lie anywhere in the graph – it need to be restricted to the reachable set of vertices.

From each edge $u \rightarrow v$ that needs to be deleted for decremental processing, we raise its weight to `MAXINT`. Also, if edge $u \rightarrow v$ was part of the previous shortest path then vertex v is pushed to the worklist with a flag specifying decremental node. If the edge was not part of the former shortest path then its weight is set to `MAXINT` but the vertex is not added to the worklist. Each vertex in the worklist looks for its next smallest predecessor by going through all the *incoming* edges to find out the new shortest distance. Therefore, decremental SSSP requires reverse edges to be maintained. The newly-found shortest distance will never be smaller than the previous distance. Once v's distance is computed then v propagates the information to all of its successors which are part of the shortest path. All the children of v which were part of the shortest paths are pushed into the worklist with flag specifying decremental node. The distance is iteratively propagated to all the levels.

Example. Consider the graph in Figure 2(b) where edge $2 \rightarrow 3$ is deleted. Since it is part of the original shortest path, we push it to the worklist with the decremental flag. Vertex 3 goes through all its incoming edges to find the next shortest path. Distance of vertex 3 is changed to 10. Now node 3 goes through all its outgoing edges and finds that node 1 had its shortest path via 3, so it pushes node 1 to the worklist with flag as decremental node. In the next iteration node 1's distance is updated to 12 from earlier 7. Since none of the edges of node 1 was part of the shortest path, no edge can be pushed to the worklist, and the algorithm terminates.

D. Fully Dynamic SSSP

Fully dynamic processing involves both the incremental and the decremental modes simultaneously. At first, for each edge $u \rightarrow v$ to be added to the graph the parent u is pushed into the worklist. Also for each edge $p \rightarrow q$ that needs to be deleted from the graph vertex q is pushed to the worklist with flag as decremental if $p \rightarrow q$ was part of the shortest path. Each vertex q in the worklist with decremental flag goes through its incoming edges to compute the new shortest path. But since the incoming edges in fully dynamic setting include incremental edges, there is a possibility of new distance to be smaller than the previous one. If the distance decreases then the vertex q is inserted in the worklist *without* the decremental flag. If the distance increases then according to the decremental approach, all the successors of q which were part of the shortest paths are pushed into the worklist with flag as decremental.

Example. Consider the graph in Figure 2(c) where edge $0 \rightarrow 1$ is newly inserted and two edges $2 \rightarrow 1$ and $2 \rightarrow 3$ are deleted. Since both the edges $2 \rightarrow 1$ and $2 \rightarrow 3$ were part of the original shortest paths, both the vertices 1 and 3 are added to the worklist with flag as decremental. Each of the worklist elements is processed in parallel, hence both the vertices compute their next nearest predecessor. Vertex 1 finds 0 as the new predecessor because of newly added edge $0 \rightarrow 1$. Also, the new distance of vertex 1 is smaller than its previous, hence node 1 gets added to the worklist without decremental flag. Vertex 3 chooses vertex 4 as the new predecessor and because of the increase in its distance, vertex 3 loops through its outgoing edges to find any successor which is part of the shortest path. Since no children are part of the shortest path from vertex 3, it does not add any vertex to the worklist. Since incremental edge $0 \rightarrow 1$ is already processed in the decremental phase, no new distances are computed by the incremental algorithm.

IV. Experimental Evaluation

We first discuss our implementation infrastructure, and then present results for incremental, decremental and fully-dynamic SSSP.

A. Implementation

We use Node package of WebCL. Node [16] is an open source environment which facilitates server-side web programming. Node uses Google V8 JavaScript to execute JavaScript applications. To improve the throughput and scalability of web applications, Node provides event driven architecture and non-blocking I/O. *npm* [17] (node package manager) enables users to install packages inside node and use them in the application. Using Node.js, JavaScript applications can be executed and debugged on the command line. node-webcl [18] is Node.js package which has implementation of WebCL. node-webcl provides fast server-side web processing for range of applications like image processing, graph algorithms, etc. node-webcl package can be installed using *npm*. We use the Node package of WebCL on a Cent-OS running Intel(R) Xeon(R) system with 32 cores, 100 GB RAM, and K40 Nvidia GPU.

For performance evaluation we have tested incremental, decremental and fully dynamic SSSP implementations against real-world graphs obtained from SNAP [19]. Table I shows characteristics of each graph. We vary the percentage of edges added or deleted from the graph and compare its running time against a static implementation. The expectation is that for smaller percentage changes, the dynamic version would outperform its static counterpart, while after a threshold, the static SSSP would perform better.

Graph	#Vertices	#Edges
Flickr	395,980	8,545,307
Rmat20	1,048,576	8,259,994
Rmat5	100,000	1,000,000
P2P	10,876	39,994

Table I
INPUT GRAPHS

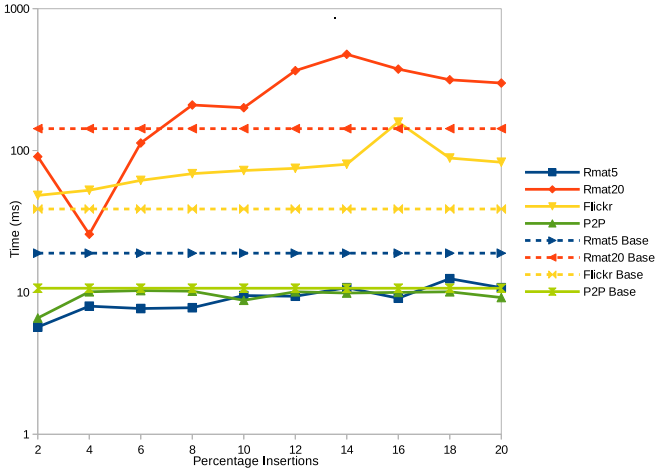

Figure 3. Incremental SSSP Performance

B. Incremental-Only and Decremental-Only Results

Performance of incremental and decremental SSSP is shown in Figures 3 and 4 respectively. We observe that the processing time increases with the percentage of edges added or deleted. Further, until a small percentage of dynamic updates, performing an incremental or decremental processing is more efficient than running SSSP all over again. This threshold differs across graphs as it is highly dependent on both the graph structure as well as the dynamic updates.

C. Fully-Dynamic Results

We combined the insertions and deletions of edges (nearly 50% each) and studied the behavior of our implementation for fully-dynamic SSSP. The results are shown in Figure 5. Similar to the incremental case, the fully dynamic version performs better for fewer updates (upto 10%). Beyond the threshold, the static version starts outperforming.

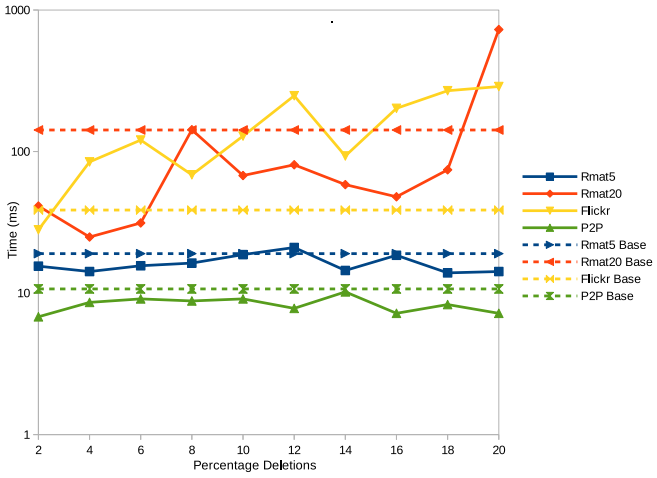

Figure 4. Decremental SSSP Performance

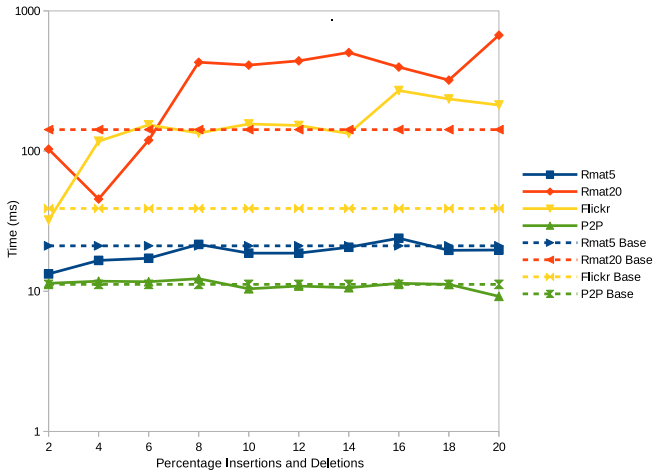

Figure 5. Fully dynamic SSSP Performance.

V. Conclusion

We presented our initial successful implementation of graph algorithms in JavaScript on GPUs. While we demonstrated it only for the shortest paths computation, we believe our investigation holds for similar graph analytic algorithms such as breadth-first search, Page Rank propagation, vertex coloring, etc. Going forward, we would like to check if other graph algorithms are amenable to GPU parallelism using JavaScript.

References

[1] "https://en.wikipedia.org/wiki/JavaScript."

[2] L. M. Surhone, M. T. Tennoe, and S. F. Henssonow, *Node.Js*. Mauritius: Betascript Publishing, 2010.

[3] "http://adambom.github.io/parallel.js/."

[4] J. Wang, N. Rubin, and S. Yalamanchili, "Paralleljs: An execution framework for javascript on heterogeneous systems," in *Proceedings of Workshop on General Purpose Processing Using GPUs*, ser. GPGPU-7. New York, NY, USA: ACM, 2014, pp. 72:72–72:80. [Online]. Available: http://doi.acm.org/10.1145/2576779.2576788

[5] S. Herhut, R. L. Hudson, T. Shpeisman, and J. Sreeram, "River trail: A path to parallelism in javascript," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '13. New York, NY, USA: ACM, 2013, pp. 729–744. [Online]. Available: http://doi.acm.org/10.1145/2509136.2509516

[6] "https://www.khronos.org/webcl/."

[7] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek, "A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene/L," in *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, ser. SC '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 25–. [Online]. Available: http://dx.doi.org/10.1109/SC.2005.4

[8] D. A. Bader and K. Madduri, "Designing Multithreaded Algorithms for Breadth-First Search and st-connectivity on the Cray MTA-2," in *Proceedings of the 2006 International Conference on Parallel Processing*, ser. ICPP '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 523–530. [Online]. Available: http://dx.doi.org/10.1109/ICPP.2006.34

[9] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew, "Optimistic parallelism requires abstractions," *SIGPLAN Not. (PLDI)*, vol. 42, no. 6, pp. 211–222, 2007. [Online]. Available: http://iss.ices.utexas.edu/Publications/Papers/PLDI2007.pdf

[10] P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the gpu using cuda," in *HiPC'07: Proceedings of the 14th international conference on High performance computing*. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 197–208.

[11] W. Hant, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, and E. Chen, "Chronos: a graph engine for temporal graph analysis," in *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 2014, p. 1.

[12] C. Ren, E. Lo, B. Kao, X. Zhu, and R. Cheng, "On querying historical evolving graph sequences," *Proceedings of the VLDB Endowment*, vol. 4, no. 11, pp. 726–737, 2011.

[13] A. Kan, J. Chan, J. Bailey, and C. Leckie, "A query based approach for mining evolving graphs," in *Proceedings of the Eighth Australasian Data Mining Conference-Volume 101*. Australian Computer Society, Inc., 2009, pp. 139–150.

[14] P. Desikan and J. Srivastava, "Mining temporally evolving graphs," in *Proceedings of the the Sixth WEBKDD Workshop in conjunction with the 10th ACM SIGKDD conference*, vol. 22, 2004.

[15] J. Sun, C. Faloutsos, S. Papadimitriou, and P. S. Yu, "Graphscope: parameter-free mining of large time-evolving graphs," in *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2007, pp. 687–696.

[16] "https://nodejs.org/en/."

[17] "https://www.npmjs.com/."

[18] "https://github.com/mikeseven/node-webcl."

[19] J. Leskovec and R. Sosič, "SNAP: A general purpose network analysis and graph mining library in C++," http://snap.stanford.edu/snap, Jun. 2014.