J. Parallel Distrib. Comput. 85 (2015) 91–103

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

All-Pairs Shortest Path algorithms for planar graph for GPU-accelerated clusters☆

Hristo Djidjev [a,*], Guillaume Chapuis [a], Rumen Andonov [b], Sunil Thulasidasan [a], Dominique Lavenier [b]

[a] Los Alamos National Laboratory, Los Alamos, NM, USA

[b] INRIA/IRISA and University of Rennes 1, Campus de Beaulieu, 35042 Rennes, France

HIGHLIGHTS

- We develop a new approach for the All-Pairs Shortest Path problem in planar graphs.
- We target execution on large CPU–GPU clusters and graphs with millions of vertices.
- We design a centralized (master/slave) and a decentralized (distributed) version.
- Our algorithms are work-efficient and allow a high-degree of parallelism.
- Our algorithms are significantly faster than the previous ones.

ARTICLE INFO

Article history:
Received 10 January 2015
Received in revised form
22 June 2015
Accepted 29 June 2015
Available online 22 July 2015

Keywords:
All-pairs shortest path problem
Planar graphs
GPGPU
Parallel computing
Floyd–Warshall algorithm
Distributed computing
Algorithm analysis

ABSTRACT

We present a new approach for solving the All-Pairs Shortest-Path (APSP) problem for planar graphs that exploits the massive on-chip parallelism available in today's Graphics Processing Units (GPUs). We describe two new algorithms based on our approach. Both algorithms use Floyd–Warshall method, have near optimal complexity in terms of the total number of operations, while their matrix-based structure is regular enough to allow for efficient parallel implementation on the GPUs. By applying a divide-and-conquer approach, we are able to make use of multi-node GPU clusters, resulting in more than an order of magnitude speedup over fastest known Dijkstra-based GPU implementation and a two-fold speedup over a parallel Dijkstra-based CPU implementation.

1. Introduction

Shortest-path computation is a fundamental problem in computer science with applications in diverse areas such as transportation, robotics, network routing, and VLSI design. The problem is to find paths of minimum weight between pairs of nodes in edge-weighted graphs, where the weight $|p|$ of a path p is defined as the sum of the weights of all edges of p. The distance between two nodes v and w is defined as the minimum weight of a path between v and w.

There are two basic versions of the shortest-path problem: in the single-source shortest-path (SSSP) version, given a source node s, the goal is to find all distances between s and the other nodes of the graph; in the all-pairs shortest-path (APSP) version, the goal is to compute the distances between all pairs of nodes in the graph. While the SSSP problem can be solved very efficiently in nearly linear time by using Dijkstra's algorithm [5], the APSP problem is much harder computationally.

Two main families of algorithms exist to solve the APSP problem exactly: the first family is based on the Floyd–Warshall algorithm [3], while the second derives from Dijkstra's algorithm. Floyd–Warshall's approach consists in iterating through every vertex v_k of the graph to improve the best known distance between every pair of vertices (v_i, v_j). The complexity of this approach is $O(|V|^3)$, where V is the set of the vertices, regardless of the density of the input graph. While the algorithm works for arbitrary graphs (including those with negative

☆ Preliminary version of this work was presented at IPDPS 2014.

* Corresponding author.
E-mail addresses: djidjev@lanl.gov (H. Djidjev), gchapuis@lanl.gov (G. Chapuis), randonov@irisa.fr (R. Andonov), sunil@lanl.gov (S. Thulasidasan), lavenier@irisa.fr (D. Lavenier).

http://dx.doi.org/10.1016/j.jpdc.2015.06.008
0743-7315/

edge weights, but no negative cycles[1]), its cubic complexity makes it inapplicable to very large graphs.

Given that Dijkstra's algorithm solves the SSSP problem, it is possible to solve the APSP problem by simply running Dijkstra's algorithm over all source vertices in the graph. When using min-priority queues, the complexity of this approach is $O(|E| + |V| \log |V|)$ for the SSSP problem, where V and E are the sets of the vertices and edges, respectively. For the APSP problem, the total complexity is thus $O(|V| * |E| + |V|^2 \log |V|)$, which becomes $O(|V|^3)$ when the graph is complete, but only $O(|V|^2 \log |V|)$ when $|E| = O(|V|)$, making this approach faster than Floyd–Warshall for sparse graphs.

Solving the All-Pairs Shortest Path problem is important not only in transportation-related problems, but also in many other domains. It is the first step to obtaining several network measures that are of importance in domains such as social network analysis or in bioinformatics. One such measure is the *betweenness centrality*, which is defined, for any vertex v, as the number of shortest paths between all pairs of vertices that pass through v, and is a measure of a v's centrality (importance) in the network. Some algorithms use the centrality of the nodes in a network in order to compute its community structure. Furthermore, in several applications, the networks that need to be analyzed may have negative weights, and hence one needs an algorithm that solves the APSP problem for graphs with real (positive as well as negative) weights. In online social networks, for instance, negative weights may be used to indicate antagonism between two individuals [13] or even conflicts and alliances between two groups [24]. Causal networks in bioinformatics also use negative edges to represent inhibitory effects [10].

In this paper, we present a new approach for solving the APSP problem for planar graphs that exploits the great degree of parallelism available in today's Graphics Processing Units (GPU). GPUs and other stream processors were originally developed for intensive media applications and thus advances in the performance and general purpose programmability of these processors have hitherto benefited applications that exhibit computational similarities to graphics applications, namely high data parallelism, high computational intensity, and data locality. However, many theoretically optimal graph algorithms exhibit few of these properties. Such algorithms often use efficient data structures storing as little redundant information as possible, resulting in highly unstructured data and un-coalesced memory access making them less-than-ideal candidates for streaming processor manipulations. Nevertheless, given the wide applicability of graph-based approaches, the massive parallelism afforded by today's graphics processors is too compelling to ignore; current GPUs support hundreds of cores per chip and even future CPUs will be manycore.

Our approach aims to exploit the structure of the input graphs and specifically their partitioning properties to parallelize shortest path computations. The approach will be especially efficient if the input graph has a good separator, which means (informally) that it can be divided into two or more equal parts removing $o(n)$ vertices or edges, where n is the number of the vertices of the graph. Such graphs are frequently seen in road networks, geometric networks and social networks [1]; all planar graphs also satisfy this property [15]. To harness the parallel computing power for solving the path problem on such graphs, we partition the input graphs into an appropriate number of parts and solve the APSP on each part and then use the partial solutions to compute the distances between all pairs of vertices in the graph. We describe two algorithms based on our approach. Our first algorithm only uses Floyd–Warshall recurrence and can therefore work with graphs with negative edges. Our second algorithm uses a Dijkstra approach for some computations and can thus only be used with positive edge weight graphs. Both algorithms have near optimal complexity in terms of the total number of operations, while their matrix-based structure is regular enough to allow for efficient parallel implementation on the GPUs. By applying a divide-and-conquer approach, we are able to make use of multi-node GPU clusters, resulting in more than an order of magnitude speedup over fastest known (Dijkstra-based) GPU implementation and a two-fold speedup over a parallel Dijkstra-based CPU implementation.

In what follows, Section 2 describes related work; in Section 3, we detail the principles of our approach; Section 4 focuses on the structure of the data and the computations and also describes how our first algorithm, master/slave model, is implemented on large multi GPU clusters. Section 5 is dedicated to another (decentralized) algorithm that allows to reduce the communications. Theoretical analysis of work and time complexity and experimental results illustrate the efficiency of the algorithms for the class of planar graphs.

2. Related work

When considering a distributed GPU implementation, both Dijkstra and Floyd–Warshall's approaches have advantages and drawbacks. Though slower for sparse graph, a Floyd–Warshall approach has the advantage of having regular data access patterns that are identical to those of a matrix multiplication. The amount of computations required for a given graph, using a Floyd–Warshall approach, solely depends on the number of vertices in the graph; therefore, balancing workloads between different processing units can be achieved easily. Dijkstra's approach is much faster for sparse graphs but, to achieve best performance, requires complex data structures which are difficult to implement efficiently on a GPU.

Implementing parallel solvers for the APSP problem is an active field of research. Harish and Narayanan [9] proposed GPU implementations of both the Dijkstra and Floyd–Warshall algorithms to solve the APSP problem and compared them to parallel CPU implementations. Both approaches however require that the whole graph fits in a single GPU's memory. They report solving APSP for a $100k$ vertex graph in around 22 min on a single GPU. A cache-efficient parallel, blocked version of the Floyd–Warshall algorithm for solving the APSP problem in GPUs is described in [12]. While the graphs mentioned in [12] are larger than what would fit onto GPU on-board memory, the largest graph instances described in the paper are still only around $10k$ vertices.

Buluç et al. [2] proposed a blocked-recursive Floyd–Warshall approach. Their implementation, running on a single GPU, shows a speedup of 17–45 when compared to a parallel CPU implementation and outperforms both GPU implementations from [9]. Their blocked-recursive implementation also requires that the entire graph fits in the GPU's global memory; therefore, they only report timings for graphs with up to $8k$ vertices. Okuyama et al. [21] proposed an improvement over the GPU implementation of Dijkstra for APSP from [9] by caching data in on-chip memory and exhibiting a higher level of parallelism. Their approach showed a speedup of 2.8–13 over Dijkstra's SSSP-based method of [9]. Matsumoto et al. [17] also proposed a blocked Floyd–Warshall algorithm that they implemented for computations on a single GPU and a multicore CPU simultaneously. Their implementation handles graphs with up to $32k$ and achieves near peak performance. Only Ortega-Arranz et al. [22] report solving APSP on large graphs — up to $1024k$ vertices. Using an SSSP-based Dijkstra approach, their implementation runs on a multicore CPU and up to 2 GPUs simultaneously. Recent experimental work on parallel algorithms for solving just

[1] Since there can be no shortest path between vertices on a negative cycle, we assume hereafter that graphs considered in this paper have no negative cycles.