All-Pairs Shortest Path Algorithms for Planar Graph for GPU-Accelerated Clusters

Hristo Djidjev, Guillaume Chapuis, Rumen Andonov, Sunil Thulasidasan, Dominique Lavenier

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*

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

☆ Preliminary version of this work was presented at IPDPS 2014

*Corresponding author

Email addresses: djidjev@lanl.gov (Hristo Djidjev), gchapuis@lanl.gov (Guillaume Chapuis), randonov@irisa.fr (Rumen Andonov), sunil@lanl.gov (Sunil Thulasidasan), lavenier@irisa.fr (Dominique Lavenier)

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 [1], 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 [2], while the second derives from Dijkstra's algorithm. The 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 edge weights, but no negative cycles [1]), its cubic complexity makes it inapplicable to very large graphs.

Given that the Dijkstra's algorithm solves the SSSP problem, it is possible to solve the APSP problem by simply running the Dijsktra'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 in many other domains. It is the first step to obtaining sev-

[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.

eral network measures that are of importance in domains such as social network analysis or in bioninformatics. 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 [3] or even conflicts and alliances between two groups [4]. Causal networks in bioinformatics also use negative edges to represent inhibitory effects [5].

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 many core.

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 sepa-

rator, 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 [6]; all planar graphs also satisfy this property [7]. 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 algorithm 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 the Floyd-Warshall and Dijkstra'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 [8] 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 fit in a single GPU's memory. They report solving APSP for a 100k vertex graph in around 22 minutes 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 [9]. While the graphs mentioned in [9] 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. [10] 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 [8]. Their blocked-recursive implementation also requires that the entire graph fit in the GPU's global memory; therefore, they only report timings for graphs with up to 8k vertices. Okuyama et al. [11] proposed an improvement over the GPU implementation of Dijkstra for APSP from [8] 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 [8]. Matsumoto et al. [12] 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. [13] 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 the SSSP problem for large graph instances using a Δ-stepping approach [14] is described in [15].

Our Contribution: We present two novel APSP algorithms and their parallel implementations to compute all shortest distances between all pairs of vertices of a graph with good partitioning properties. To make the algorithms scalable to large graphs, our implementations use a combination of shared and distributed-memory GPU computing; the current implementations target executions on large clusters of GPUs in order to handle graphs with up to a million of vertices. The results show that the trillion shortest distances of a million vertex graph can be found in less than 25 minutes using 64 cluster nodes with 2 GPUs each with our first implementation and under 6 minutes using 256 such cluster nodes with our second implementation.

We view our contributions in the following:

(i) We develop a new approach that is simultaneously work-efficient, has a high-degree of parallelism, and is built upon matrix operations; we are aware of no previous APSP approach with such properties.

(ii) Our approach is based on Floyd-Warshall algorithm and uses massive parallelism; both fine-grained at GPU level as well as coarse-grained employing upto 512 GPUs.

(iii) We design and implement two algorithms based on our approach: a centralized (master/slave type) and a decentralized (communications reducing) one. While the former has already been presented in the conference version of this paper [16], the latter algorithm is firstly described here.

(iv) The numerical results show that our algorithms outperform the previous algorithms by orders of magnitude with respect to running times using the same or similar computational resources.

(v) The master/slave approach is entirely based on the Floyd-Warshall recurrence and has the advantage to work with arbitrary–negative as well as positive–weights.

(vi) The matrix structure of our approach will allow it to get additional efficiency boost from any pipelined vector features not available in current GPUs.

3. Algorithm details

In this section, we give the overall structure and the idea of the approach and describe its individual steps, but without discussing details of the GPU implementation. We start with an overview and then give details on each of its steps.

3.1. Overview

Our algorithm takes as input a weighted directed or undirected graph G with n vertices and computes the distances between all pairs of vertices of G. We currently do not output routing information, which can be used to reconstruct the shortest paths, but computing such an information requires a minor modification in the algorithm and would increase the run times and memory requirements by at most a constant factor of two.

Our algorithm is based on a divide-and-conquer approach and consists of four steps (see Algorithm 1). In the first step, the original graph G is partitioned into k components of roughly equal sizes using a min-cut like heuristic – our implementation uses a k-way partitioning method from the METIS library [17]. In the second step, the APSP problem is solved on each component independently; in the third step the distance information computed for the components is used to compute distances between all pairs of boundary vertices of G (a *boundary* vertex is one that is adjacent to a vertex from another component); and in the final step the information obtained in steps two and three is combined to compute shortest paths between non-boundary pairs of vertices of G. We will use

the following notation: $\text{dist}_i(v, w)$ will denote the (approximate) value of the distance between v and w computed in Step i, for $i = 2, 3, 4$, and $\text{dist}_G(v, w)$ will denote the (exact) distance in G. Next we will describe the steps in more detail.

3.2. Step 1: Graph decomposition

In Step 1, the input graph G is divided into k components of roughly equal sizes. The decomposition is done by identifying a set of edges (a *cut* set) whose removal from G results into a disconnected graph of k parts we call *components*. The set of all components is called a *partition*. Note that while by the standard definition in graph theory a component is connected, this is not a requirement in our case (although in the typical case our components will be connected). A requirement is that every vertex in G belongs to exactly one component of the partition. Moreover, in order for the resulting APSP algorithm to be efficient, the cut set of edges should be small. Not all classes of graphs have such partitions, but some important classes do. These include the class of planar graphs, the class of graphs of low genus (that can be drawn on a sphere with a small number of "handles" [18]), some geometric graphs, and graphs corresponding to networks with good community structure.

3.3. Step 2: Computing distances within each graph component

Step 2 involves computing the distances in each component of the partition $\mathcal{P}$ of G using a conventional algorithm, e.g., the Floyd-Warshall's or Dijkstra's algorithm. For each component $C \in \mathcal{P}$ and any two vertices s and t of C, the output of this step is the minimum length of a path between s and t that is restricted to lie entirely in C. Hence, the distance computed between s and t may be larger that the distances between s and t in G, if there is a shorter path between them that goes out of C. Nevertheless, as we will show in the next subsections, the computed approximate distances can be used to efficiently compute the accurate distances in G.

Algorithm 1 Partitioned All-Pairs Shortest Path algorithm

```
PROCEDURE compute_apsp(G)
INPUT: A planar graph G(V,E), where V is a set of vertices and E a
    set of weighted edges between these vertices.
OUTPUT: The distance of the shortest path between any two pairs of
    vertices in G.

  // Step 1
  Partition G into k roughly equal components using METIS

  //Step 2
  for each Component C in G
      Floyd-Warshall(C) % Compute APSP(C) by Floyd-Warshall
  end for

  // Step 3
  Graph BG = extract_boundary_graph(G)
  compute_apsp(BG) % apply the algorithm recursively
  for each Component C in G
    Floyd-Warshall(C) % Compute APSP(C) by Floyd-Warshall
  end for

  // Step 4
  for each Component C1 in G
    for each Component C2 in G
      compute_apsp_between_components(C1, C2)
    end for
  end for
```

In order to implement this step, for each component $C \in \mathcal{P}$, a subgraph is extracted containing vertices from the current component and existing edges between these vertices. Any APSP algorithm can then be applied in order to compute distances in each of these sub-graphs. This step thus has k independent tasks – one for each sub-graph – that can be computed in parallel. Since each component contains roughly n/k vertices, using an algorithm whose complexity solely depends on the number of vertices allows these tasks to be computed in roughly the same number of operations. This property can be advantageous depending on the type of parallelism that we want to exploit.

3.4. Step 3: Computing distances in the boundary graph

In step 3, we first extract the *boundary graph* BG of G with respect to the partition $\mathcal{P}$. The vertices of BG are defined to be all boundary vertices of G. There are two types of edges of BG. The first type of edges are edges in G between boundary vertices from different components. The weights on these edges are the same as their weights in G. The second type of edges, which we call *virtual* edges, are between boundary vertices in the same components – for any two boundary vertices v and w belonging to the same component C there is an edge (v, w) in BG with weight equal to the distance between v and w computed in Step 2. Hence, BG is a compressed version of the original graph, where all non-boundary vertices have been removed, and instead of them shortest path information encoded in the weights of the new edges of BG. Having constructed BG, we then solve for it the APSP problem using a conventional APSP algorithm.

Despite the fact that the distances encoded in the weights of the new edges of BG are only approximate, the distances between the boundary nodes of BG computed at the end of Step 3 are exact. The next lemma formally establishes this fact.

Lemma 1. *For any two boundary vertices v and w, the distance between v and w in BG is equal to their distance in G.*

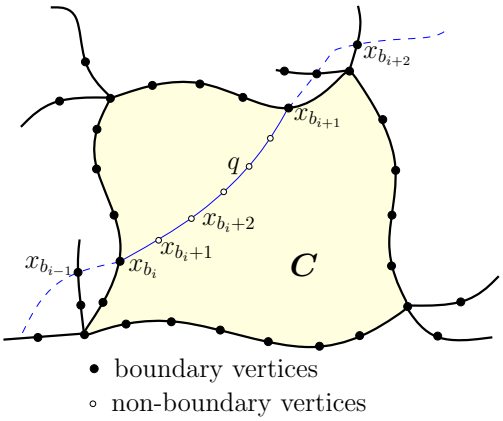

Figure 1: Illustration to the proof of Lemma 1. The region inside the bold-line loop is a component C with the subpath $q = (x_{b_i}, x_{b_i+1}, \ldots, x_{b_{i+1}})$ of p inside it.

Proof. Let $p = (v = x_1, x_2, \ldots, x_l = w)$ be a shortest path in G and let $(x_{b_1}, x_{b_2}, \ldots, x_{b_j})$ be the subsequence of all boundary vertices in p, i.e., $1 = b_1 < \cdots < b_j = l$ and there are no boundary vertices on p between x_{b_i} and $x_{b_{i+1}}$. Hence $p' = (x_{b_1}, x_{b_2}, \ldots, x_{b_j})$ is a path in BG. We are going to estimate the length of p'.

Let $h = (x_{b_i}, x_{b_{i+1}})$ be an edge of p'. If x_{b_i} and $x_{b_{i+1}}$ are from different components, then, by the definition of BG, h is also an edge of G with the same weight as in BG. If x_{b_i} and $x_{b_{i+1}}$ are from the same component C (Figure 1), then h corresponds to a subpath $q = (x_{b_i}, x_{b_i+1}, \ldots, x_{b_{i+1}})$ of p consisting of vertices from only C, by the assumption that p' contains all the boundary vertices of p. Hence, the weight of h and the length of q are the same. By induction on the number of the edges of p', p and p' have the same length, which implies that the distance between v and w in BG is no greater than the distance between them in G. The reverse inequality is obtained in the same way, namely, by showing that any path in BG can be transformed into a path of the same length in G by replacing each virtual edge of the former with the corresponding shortest path computed in Step 2. The claim follows. □

This step presents no apparent parallelism, since only one task needs to be

computed. This absence of parallelism at this step may be a major bottleneck for a coarse-grain parallel implementation as boundary graphs can be very large. This issue can however be mitigated by applying our current algorithm recursively on the boundary graph. Boundary graphs are nevertheless denser than the original graph with the addition of virtual edges at Step 2. Boundary graphs are therefore less easily partitioned than input graphs - the number of edges cut per node for a given number of components will be higher.

3.5. Step 4: Distances between non-boundary vertices

In Step 4 we compute distances where at least one vertex is non-boundary using the information computed in Steps 2 and 3. In order to compute the distance between two non-boundary vertices v_i and v_j from (not necessarily different) components C_i and C_j respectively, we need to find boundary vertices b_i and b_j from components C_i and C_j, respectively, that minimize the sum $\text{dist}_2(v_i, b_i) + \text{dist}_3(b_i, b_j) + \text{dist}_2(b_j, v_j)$, where dist_2 and dist_3 are the distances computed in Step 2 and Step 3, respectively. By our analysis above, dist_3 is the same as the distance in G, but dist_2 is not. We need therefore to prove that such a method produces accurate distances in G.

Lemma 2. *Let v_i and v_j be two vertices from different components C_i and C_j, respectively. Define $B_i = C_i \cap BG$, $B_j = C_j \cap BG$, and*

$$dist_4(v_i, v_j) = \min\{dist_2(v_i, b_i) + dist_3(b_i, b_j) + dist_2(b_j, v_j) \mid b_i \in B_i, b_j \in B_j\}. \quad (1)$$

Then $dist_4(v_i, v_j)$ is equal to the distance in G between v_i and v_j.

Proof. Let p be a shortest path in G between v_i and v_j. Since v_i and v_j belong to different components, then p will contain at least one vertex from B_i and at least one vertex from B_j. Let b_i be the first vertex on p from B_i and b_j be the last vertex on B_j (Figure 2). Let p_1 be the portion of p between v_i and b_i, p_2 be the portion between b_i and b_j, and p_3 – the portion between b_j and v_j. Since any subpath of a shortest path is also a shortest path between the corresponding

endpoints, p_1 is a shortest path in G between v_i and b_i, i.e., $|p_1| = \text{dist}_G(v_i, b_i)$. Moreover, by the definition of b_i as the first boundary point of C_i on p, p_1 is entirely in C_i and hence $|p_1| = \text{dist}_2(v_i, b_i)$. In the same way one can prove that $|p_2| = \text{dist}_2(b_j, v_j)$. Finally, $|p_3| = \text{dist}_G(b_i, b_j) = \text{dist}_3(b_i, b_j)$ by Lemma 1. Hence

$$|p| = |p_1| + |p_2| + |p_3| = \text{dist}_2(v_i, b_i) + \text{dist}_3(b_i, b_j) + \text{dist}_2(b_j, v_j).$$

By the definition of $\text{dist}_4(v_i, v_j)$ as a minimum over all $b_i \in B_i, b_j \in B_j$, the last equality implies $\text{dist}_4(v_i, v_j) \leq \text{dist}_G(v_i, v_j)$. But since $\text{dist}_4(v_i, v_j)$ is a length of a path between v_i and v_j, while $\text{dist}_G(v_i, v_j)$ is the length of a shortest path, then $\text{dist}_4(v_i, v_j) \geq \text{dist}_G(v_i, v_j)$. Combining the last two inequalities we infer that none of them can be a strict inequality, i.e., $\text{dist}_4(v_i, v_j) = \text{dist}_G(v_i, v_j)$. □

Lemma 3. *Let v_i and v_j be two vertices from component C_i. Then $dist_G(v_i, v_j) = \min\{dist_2(v_i, v_j), dist_4(v_i, v_j)\}$, where $dist_4$ is as defined in Lemma 2.*

Proof. Consider the following two cases. If p leaves C_i, then p should cross the boundary B_i at least twice. Define b_i and b_j as the first and last vertex from B_i on p. Then exactly the same arguments as in Lemma 2 apply to the three paths into which b_i and b_j divide p. In this case $\text{dist}_G(v_i, v_j) = \text{dist}_4(v_i, v_j)$. If p does not leave p, then Step 2 will compute the accurate distance in G between v_i and v_j, and therefore $\text{dist}_G(v_i, v_j) = \text{dist}_2(v_i, v_j)$. □

The lemmas imply that the distances in G between all pairs of vertices where at least one of the vertices is non-boundary can be computed by using (1). Since we don't know which pair (b_i, b_j) of boundary nodes corresponds to the minimum in (1), we have to try all such pairs, resulting in total of $|B_i||B_j|$ operations needed for computing $\text{dist}_G(v_i, v_j)$. For a graph with k components, we need to compute the distances between pairs in any pair of components; we therefore have k^2 independent tasks. Components being of roughly equal sizes, these tasks also represent the same amount of computations. This step is the most computationally intensive, but presents massive, already balanced, coarse-grain parallelism.

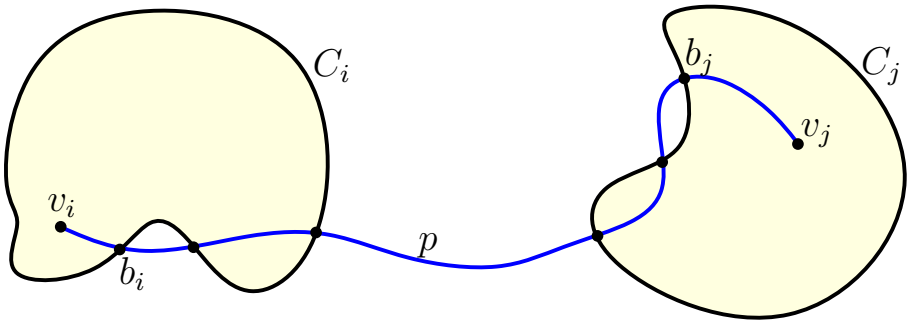

Figure 2: Illustration to the proof of Lemma 2. Note that while in the figure both v_i and v_j are non-boundary, the proof does not make such an assumption.

4. Master/slave approach: data organization, implementation details and results

In this section, we describe our implementation of Algorithm 1. Except for using METIS library for graph partitioning, the matrix multiplication routines from [19], which we adapted to our operators and to non-power of two matrices and the block-recursive matrix multiplication from [10] also adapted to non-power of two matrices, all the other code was written from scratch. We first focus on how operations described in the previous section translate in terms of data structures. We then detail the two-level parallel aspect of our implementation. We finally describe the current main memory bottleneck of this approach.

4.1. Data organization

A simple way to represent a weighted graph is to use an adjacency matrix. For very large graphs however, such a memory intensive representation is often avoided. Instead, large sparse graphs are stored using lists; sub-matrices, corresponding to sub-graphs, are extracted from these lists. For simplicity reasons, we can however assume that a large adjacency matrix representation is available and keep in mind that sub-matrix extraction operations are slightly more costly than they appear. We are also taking into account the fact that, even when

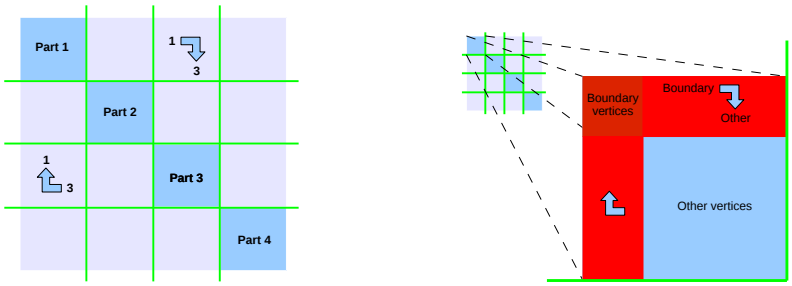

Figure 3: Adjacency matrix after reordering of the vertices. Vertices from the same component are stored contiguously starting with boundary vertices (in red or dark grey if printed in b&w).

the input graph (matrix) is sparse, the output is always a dense matrix as it encodes the distances between all pairs of vertices.

Partitioning the graph is performed using a k-way partitioning routine from the METIS library [17]. The result is a partitioning of the graph into k parts such that the number of edges with endpoints in different parts is minimized. Since that partitioning problem is NP-hard, METIS computes an approximation based on heuristics. Vertices are then reordered so that vertices belonging to the same component are numbered consecutively starting with the boundary vertices – see Figure 3.

Diagonal sub-matrices contain information about sub-graphs for each component; non-diagonal sub-matrices contain known shortest distances between components. Within each diagonal sub-matrix, the top left sub-matrix contains information about the sub-graph induced by boundary vertices of the component; the bottom right sub-matrix contains information about the sub-graph induced by non-boundary vertices of the component and the rest of the diagonal sub-matrix contains known shortest distances between boundary and non-boundary vertices.

For Step 2, diagonal sub-matrices are extracted; a Floyd-Warshall approach is then used to compute shortest distances. The Floyd-Warshall algorithm guarantees that the total number of operations for a single matrix solely depends on the size of the matrix. Since all components of the graph have roughly the

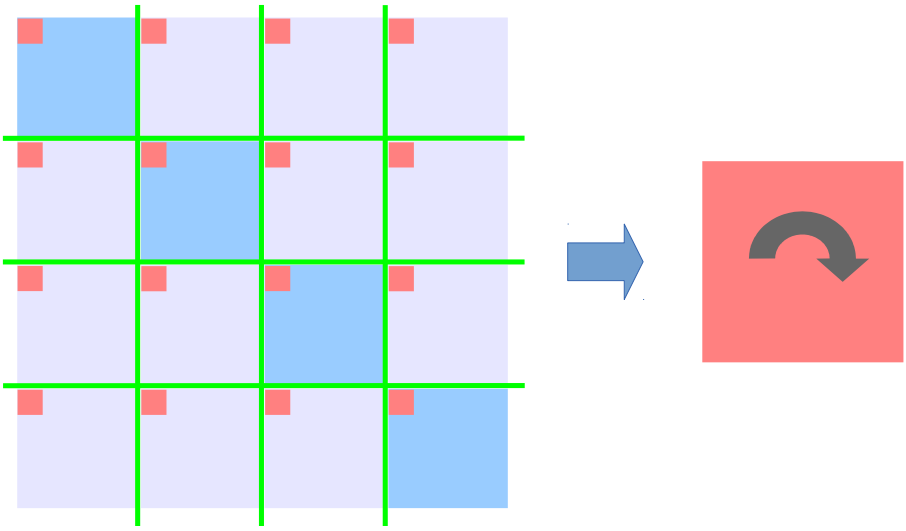

Figure 4: The boundary matrix, here in red (or dark grey if b&w), is scattered over the adjacencymatrix. Step 3 consits in reconstituting the boundary matrix and computing shortest distances.

same number of vertices, all diagonal sub-matrices represent roughly the same amount of operations.

For Step 3, the boundary matrix is extracted – see Figure 4. We then apply the same algorithm recursively reducing the number k of component at each iteration. Recursion in the current implementation stops when $k = 1$; alternatively, the stopping condition can be changed to checking if the boundary graph BG is so dense that it does not have good partitioning (in terms of the number b of boundary vertices of BG), e.g., if b is greater than half the number of the vertices of BG. At that point the APSP subproblem is solved using Floyd-Warshall.

For Step 4, we compute shortest distances between every pair of distinct components. This process corresponds to filling non-diagonal sub-matrices. For two components I and J, filling the associated, I to J, non-diagonal sub-matrix requires information from three sub-matrices:

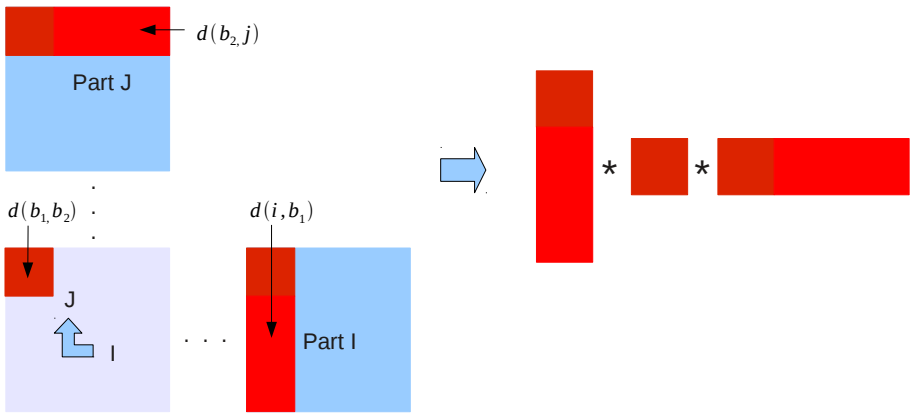

Figure 5: Computations associated to each non-diagonal sub-matrix uses data from 2 diagonal sub-matrices and part of the non-diagonal sub-matrix itself. Computations are similar to matrix multiplications.

- the non-diagonal sub-matrix being filled. We are particularly interested in the part of the sub-matrix containing shortest distances between boundary vertices from component I to boundary vertices from component J.
- the diagonal sub-matrix corresponding to component I - located in the same row as the non-diagonal sub-matrix being filled. We are particularly interested in the part of this diagonal sub-matrix that contains shortest distances from any vertex of component I to boundary vertices.
- the diagonal sub-matrix corresponding to component J - located in the same column as the non-diagonal sub-matrix being filled. We are particularly interested in the part of this diagonal sub-matrix that contains shortest distances from boundary vertex of component J to any vertex - see left of Figure 5.

Shortest distances from vertices from component I to vertices from component J are obtained by multiplying the three parts of sub-matrices - as shown on the right of Figure 5 - where $(+, *)$ operations are replaced with $(min, +)$ operations.

4.2. Work analysis

Next, we will try to estimate the work (number of operations) of the algorithm. Since the work depends on the partitioning properties of the input graph, we will do the analysis for the case of planar bounded-degree graphs. For that class of graphs, there exists a partitioning of any n-vertex graph into k roughly equal parts such that the number of boundary vertices in each part is $O(\sqrt{n/k})$ [20]. We make the assumption that METIS produces a partition with such properties. Although the partition METIS does not come with theoretically guaranteed bounds, it works in practice better than alternative algorithms that have such guarantees, which is the reason we chose it. The time needed for Step 1 is

$$O(n \log n). \tag{2}$$

In Step 2, we have k sub-tasks of computing APSP on graphs of size $O(n/k)$ using an algorithm of cubic complexity, so the number of operations for that step is

$$O(k(n/k)^3) = O(n^3/k^2). \tag{3}$$

In Step 3, we have to solve the APSP on a graph of size $O(k\sqrt{n/k}) = O(\sqrt{kn})$. Using an algorithm with complexity $O(N^\alpha)$, where N is the number of the vertices of the subgraph, the number of operations for this step is

$$O(N^\alpha) = O((kn)^{\alpha/2}). \tag{4}$$

For Step 4, we have $k^2 - k$ independent tasks and each task involves the multiplication of three matrices with dimensions $n/k \times \sqrt{n/k}$, $\sqrt{n/k} \times \sqrt{n/k}$, and $\sqrt{n/k} \times n/k$, respectively (see Figure 5). Computing the product of the first and the second matrix takes $O((n/k)\sqrt{n/k}\sqrt{n/k}) = O((n/k)^2)$ operations and finding the product of the resulting $n/k \times \sqrt{n/k}$ matrix and the third matrix takes $O((n/k)\sqrt{n/k}(n/k)) = O((n/k)^{5/2})$ operations, which is the dominating term. Hence, the total number of operations for Step 4 is

$$O(k^2(n/k)^{5/2}) = O(n^{5/2}/k^{1/2}) \tag{5}$$

The total number of operations is the sum of the numbers computed for Steps 1, 2, 3, and 4 and is minimized when $(kn)^{\alpha/2} = n^{5/2}/k^{1/2}$ or $k^{\alpha+1} = n^{5-\alpha}$. If in Step 3 Floyd-Warshall is used, then $\alpha = 3$ and $k = n^{1/2}$ is optimal, resulting in a bound of $O(n^{9/4})$ for the total number of operations, slightly worse than the theoretical lower bound of $O(n^2)$ (the output complexity of the algorithm). Our implementation in fact uses recursion in Step 3 so the total complexity is even closer to quadratic, but we will skip the details of the exact evaluation since the analysis gets much more complex.

4.3. Memory analysis

For very large input graphs, memory usage becomes an issue. As stated previously, an entire adjacency matrix for the graph cannot be allocated; the graph is instead kept in memory as a list of edges, a much more memory-efficient representation. Even with this efficient representation, temporary sub-matrices need to be kept in memory: diagonal sub-matrices and boundary matrices. When recursively computing Step 3, boundary matrices are output to files so as to only keep a single boundary matrix in memory.

Final results for diagonal sub-matrices are only obtained at the end of Step 3. As soon as final values for these diagonal sub-matrices are obtained, they are output to files; only relevant parts are kept in memory for Step 4; namely, parts of these sub-matrices containing shortest distances from and to boundary vertices. Distances between non-boundary vertices are thus discarded from main memory at the end of Step 3.

In the centralized approach, the master needs to keep the k diagonal sub-matrices of size $\frac{n}{k} \times \frac{n}{k}$ plus the initial boundary matrix of size $\sqrt{kn} \times \sqrt{kn}$. The total master memory requirement is therefore

$$k(n/k)^2 + (\sqrt{kn})^2 = n^2/k + kn. \tag{6}$$

The above is the current limiting factor in terms of memory usage. Section 4.6 discusses ways to overcome this limitation. It is however probable that

prohibitive run-times or an amount of results too large to process may become the limiting factor before main memory usage does.

4.4. Parallel implementation

Our implementation specifically targets large clusters of hybrid systems - possessing both a multicore CPU and manycore GPUs. This implementation exploits parallelism at two levels. At a coarse-grain level, large independent tasks - corresponding to computations of diagonal and non-diagonal sub-matrices - can be performed simultaneously on different nodes of a cluster. At a fine-grain level, each task is computed on a massively parallel GPU. Remaining CPU cores handle tasks that are not suited for GPUs: input/output file operations and communication with other nodes.

Coarse-grain parallelism

Steps 2 and 4 of our algorithm exhibit interesting parallel properties: a large number of balanced, independent tasks; k tasks for Step 2 and $k^2 - k$ for Step 4. Using the MPI standard [21], these tasks are distributed across nodes of the cluster for simultaneous computations. One master node is in charge of reading the input graph file, calling the partitioning routine and sending tasks to a number of slave nodes equal to the number of available GPUs on the cluster. Depending on the cluster's topology, the number of master and slave nodes will not match the number of physical nodes used on the cluster if each cluster node contains more than one GPU.

For Step 3, the large initial boundary matrix is computed recursively using the same algorithm with decreasing values for the number k of components. The amount of independent tasks therefore decreases with k, until a single, smaller boundary matrix is obtained and computed by a single slave node.

Fine grain parallelism

Upon receiving a task from the master node, each slave node then sends the corresponding data to its GPU for computations, retrieves results and send them back to the master node. Tasks are of two different kinds: diagonal workloads,

which consist in computing shortest distances over a small subgraph, and non-diagonal workloads, which consist in multiplying three matrices.

Computations of diagonal workloads are implemented on the GPU using a blocked-recursive Floyd-Warshall approach developed by [10] and adapted for non-power of 2 matrices. Non-diagonal workloads require less synchronization and can be implemented using a fast matrix-multiplication approach derived from [22] and adapted for $(min, +)$ operations.

In this configuration, each physical node on the cluster makes use of as many CPU cores as there are available GPUs. If more CPU cores are available than GPUs, extra cores are used for I/O operations and MPI communication. On slave nodes, remaining CPU cores are used for outputting final results to disk. On large clusters, communication between the master node and slave nodes can become a bottleneck, leaving slave nodes idle while waiting for the master node to be available. In order to increase the availability of the master node, a single CPU thread is used to initiate communications with slave nodes while remaining CPU cores handle the rest of the communications, updating data structures with temporary results and outputting final results to disk.

4.5. Time analysis

As already indicated, Step 1 is sequential and its time is $O(n \log n)$. Since it is of a lower order of magnitude than the other steps, we will ignore it in further analysis.

Let p be the number of nodes. Step 2 consists in performing k independent tasks – each consisting in the execution of the Floyd-Warshall algorithm over a graph of size n/k. The master is in charge of communicatind to the slaves the k transfers – each one them consists of a sub-matrix of size $(n/k \times n/k)$. The communication time corresponding to this data transfer is hence $O(n^2/k)$. Once the data is received, the respective node starts executing the Floyd-Warshall algorithm. Since k independent tasks are executed by p nodes simultaneously, the corresponding computation time is $O(n^3/(pk^2))$ if $p \leq k$, and for the total

time for Step 2 we obtain

$$O\left(\frac{n^2}{k} + \frac{n^3}{pk^2}\right), \text{ if } p \leq k. \tag{7}$$

For the time for Step 3 (when run on G), we have to estimate the time it will take to run Algorithm 1 on BG. We will actually estimate the work (thereby assuming no parallelism), since the time for Step 3 will be dominated by the time for other steps of the algorithm. We cannot directly use the work bound derived in Section 4.2 since BG is not planar, but we can use the same methodology. First, we need to estimate the size of the boundary graph for a k'-partition of BG. We use the fact that removing all the new (virtual) edges between boundary vertices in the same component results in a planar graph of $N = \sqrt{nk}$ vertices and with faces of size $f = O(\sqrt{n/k})$. Using the simple-cycle separator theorem [23], which states that that planar graph has a k'-partition of size $O(\sqrt{Nfk'})$, we find there is a k'-partition of BG with

$$O(\sqrt{Nfk'}) = O(\sqrt{\sqrt{nk}\sqrt{n/k}\,k'} = O(\sqrt{nk'}) \tag{8}$$

boundary vertices.

Similarly to the derivation of the work formula in Section 4.2, the optimal value for k' has to be chosen such as to balance the times for Step 3 and Step 4 on BG, which are $O((nk')^{3/2})$ ((8) and (4) for $\alpha = 3$) and $O((nk)^{5/4}/k'^{1/2})$ (by (5)), respectively. Hence the optimal value of k' (within a constant factor) can be determined by solving the equation

$$(nk')^{3/2} = (nk)^{5/4}/k'^{1/2},$$

which gives $k' = k^{5/8}/n^{1/8}$. Replacing that value in either term of the previous equation gives an estimation for the time for Step 3 as

$$O(n^{\frac{21}{16}}k^{\frac{15}{16}}). \tag{9}$$

Note that (9) can be used instead of (4) in the work analysis, leading to a small improvement of the work bound.

For Step 4 (for G), we have $k^2 - k$ independent tasks and each task involves the multiplication of three matrices with dimensions $n/k \times \sqrt{n/k}$, $\sqrt{n/k} \times \sqrt{n/k}$, and $\sqrt{n/k} \times n/k$, respectively. The amount of transmitted data from the master to a given slave is proportional to $n/k + 2(n/k)^{3/2} = O((n/k)^{3/2})$.

On the other hand, as shown in (5), the number of operations is $(n/k)^{5/2}$. The computation time needed by p nodes to compute the $k^2 - k$ independent tasks is hence given by $O((k^2/p)(n/k)^{5/2})$, while for the total time of Step 4 we obtain, by adding the data transmission time to the computation time

$$O\left((k^2/p)(n/k)^{5/2} + (n/k)^{3/2}\right), \text{for } p \leq k^2 - k. \tag{10}$$

Summing up equations (7), (9) and (10), we obtain for the total time of the centralized algorithm

$$O\left(\frac{n^2}{k} + \frac{n^3}{pk^2} + n^{\frac{21}{16}} k^{\frac{15}{16}} + \left(\frac{n^{\frac{5}{2}}}{pk^{\frac{1}{2}}}\right) + \left(\frac{n}{k}\right)^{\frac{3}{2}}\right). \tag{11}$$

We want to further analyze formula (11) in order to find which values of p and k result in minimum time in asymptotic terms. For this end, we substitute $k = n^\alpha$ and $p = n^\beta$. For each fixed pair of values for α and β, the asymptotic time (11) is dominated by one of its five terms. Hence, we want to find α and β that minimize

$$\max\{n^{2-\alpha}, n^{3-2\alpha-\beta}, n^{21/16+15/16\alpha}, n^{5/2-\alpha/2-\beta}, n^{3/2(1-\alpha)}\}.$$

Taking into account that logarithm is an increasing function, the above problem is equivalent to finding α and β that minimize

$$\max\{2-\alpha, 3-2\alpha-\beta, 21/16+15/16\alpha, 5/2-\alpha/2-\beta, 3/2(1-\alpha)\}. \tag{12}$$

We can transform (12) into the linear program:

Minimize t

subject to:

$$\begin{aligned} t &\geq 2 - \alpha \\ t &\geq 3 - 2\alpha - \beta \\ t &\geq 21/16 + 15/16\alpha \\ t &\geq 5/2 - \alpha/2 - \beta \\ t &\geq 3/2(1 - \alpha)\,. \end{aligned} \tag{13}$$

Using a linear solver, we find a solution $\alpha = 0.35$, $\beta = 0.68$ and the corresponding value of the objective $t = 1.65$. These correspond to values of the parameters $k = n^{0.35}$, $p = n^{0.68}$, and corresponding value for the time $T = n^{1.65}$, which is the smallest possible time even if using infinite number of processors.

We can modify the linear program (13) so that, for different fixed values of β (resp. p), it will compute the best corresponding values of α (resp. k). Then we can plot the dependence between p and the best values for k and T.

We ran the solver (we used Cbc) for $\beta = 0.05, 0.1, 0.15, \ldots, 1$. The corresponding optimal values for k and T are illustrated on Figure 6.

4.6. Results

In this section, we compare our implementation to two parallel Dijkstra implementations. It is important to note that our implementation allows graphs with negative edges – but no negative cycles – unlike Dijkstra-based approaches.

In order to test our implementation, we generated random graphs with increasing numbers of vertices, ranging from 1024 to 1024k. These graphs, generated using the LEDA library [24], were made planar to ensure good partitioning properties.

Computations were run on a cluster of more than 300 computer nodes; each node is equipped with two NVIDIA C2090 GPUs, a 16 core Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz and 32 GB of RAM.

Our implementation handles instances up to 512k vertices without using external memory. For the very last instance, the use of external memory was

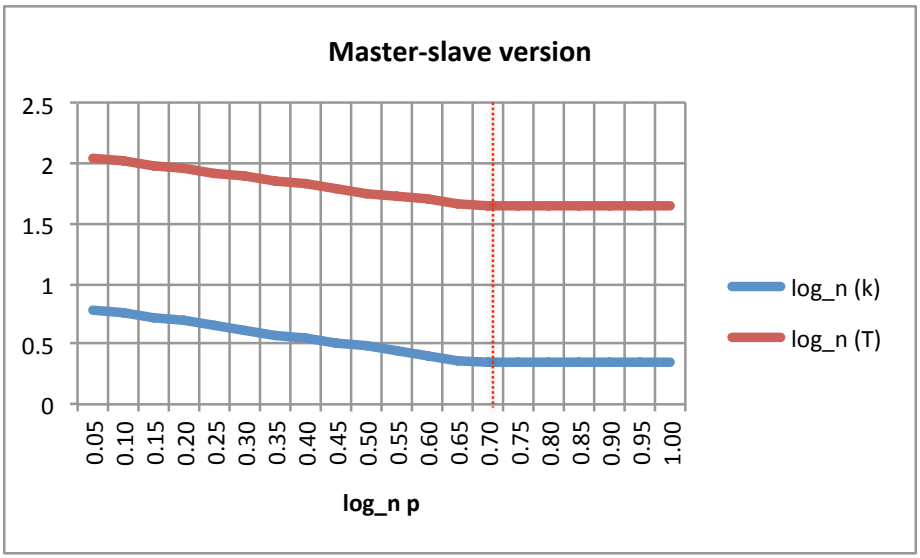

Figure 6: Optimal values of k and T as functions of the number of processors p, in a log-log scale. The red line shows the value of p beyond which there is no speedup.

required to fit in the 32 GB of main memory. We later refer to our implementation without using external memory as "Part. APSP no EM" and our implementation using external memory as "Part. APSP EM".

The GPU Dijkstra implementation from [13] is, to the best of our knowledge, the only implementation that was reported to solve APSP for graphs with up to 1024k vertices; we later refer to this implementation as "GPU Dijkstra". This implementation parallelizes SSSP computations on a single computer using two GPUs and a multicore CPU. In order to compare this implementation to ours, we restricted computations of both implementations to using only two GPUs. Both implementations could therefore run on a single cluster node; no communication between nodes were therefore required.

Figure 7 shows the runtimes for GPU Dijkstra and Part. APSP EM for graphs with numbers of vertices ranging from 1024 to 1024k using only two GPUs. GPU Dikstra could not compute the last two instances - 512k and 1024k vertices - within the 10 hour limit enforced on the cluster. We can see

Figure 7: Evolution of run times with respect to the number of vertices. Two implementations are compared: our implementation using external memory and the GPU Dijkstra implementation from [13]. Computations were run using two GPUs on a single cluster node.

that our implementation is significantly faster than GPU Dijkstra.

Figure 8 shows the evolution of the speedup of our method without using external memory with respect to the number of GPUs used for the computations. Speedups are calculated using the run time obtained using only one GPU as a reference. Computations were done for the 512k vertex instance using the Partitioned APSP with no need for external memory storage. We can see that coarse-grain parallelism is close to optimal up to around 31 GPUs; almost no benefit can however be gained from using more than about 63 GPUs. The reason for this stagnation of the speedup above 63 GPUs is the saturation of communication with the master node.

The scalability can be improved using a coarse-grain parallelism approach that would relieve the master node of some of its communication. This issue could for instance be addressed by creating a hierarchy of master nodes; some computations would be redundant between the different master nodes - handling the main data structure - but this would only represent a negligible fraction of the overall workload. Another approach - already implemented and presented

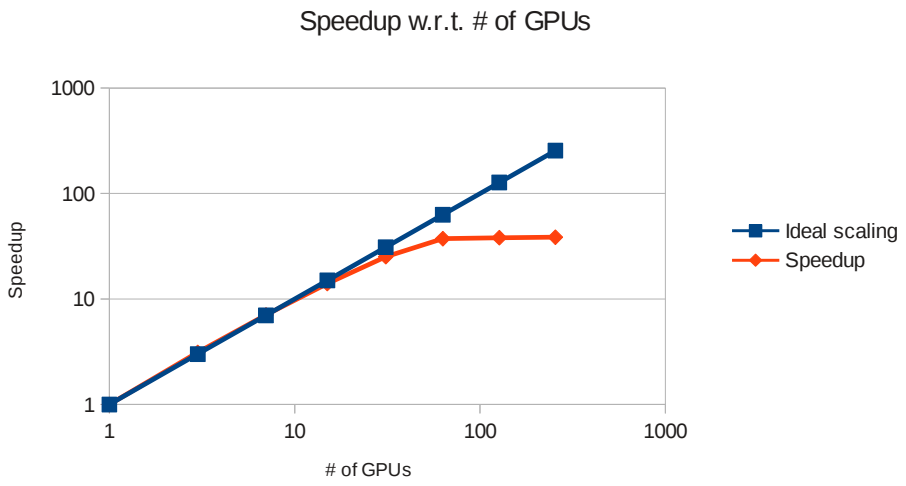

Figure 8: Evolution of speedups with respect to the number of GPUs. The ideal scaling line is given as a reference.

in section 5 - consists in removing entirely the hierarchy between nodes and having them work in parallel on statically attributed tasks.

Figure 9 shows a comparison between our two implementations and a distributed Dijkstra approach - later referred to as CPU Dijkstra - for graphs ranging from 1024 to 1024k vertices. The distributed Dijkstra approach was implemented by dynamically distributing SSSP computations for each vertex of the graph over every core of every available cluster node. The Dijkstra-based implementation used is that of the Boost C++ library [25]. This experiment is not intended to compare directly the performances of 2 GPUs versus a multi-core CPU. Instead, we intend to show that our approach is competitive with a distributed Dijkstra approach given a fixed number of hybrid cluster nodes. The run times presented in Figure 9 were obtained using 64 cluster nodes, with CPU Dijkstra running on 64 CPUs and our partitioned algorithms running on 128 GPUs. We can see that our version using external memory obtains very similar run times to that of the distributed Dijkstra version, while allowing graphs with negative edge weights to be handled. Our version without external memory is however significantly faster.

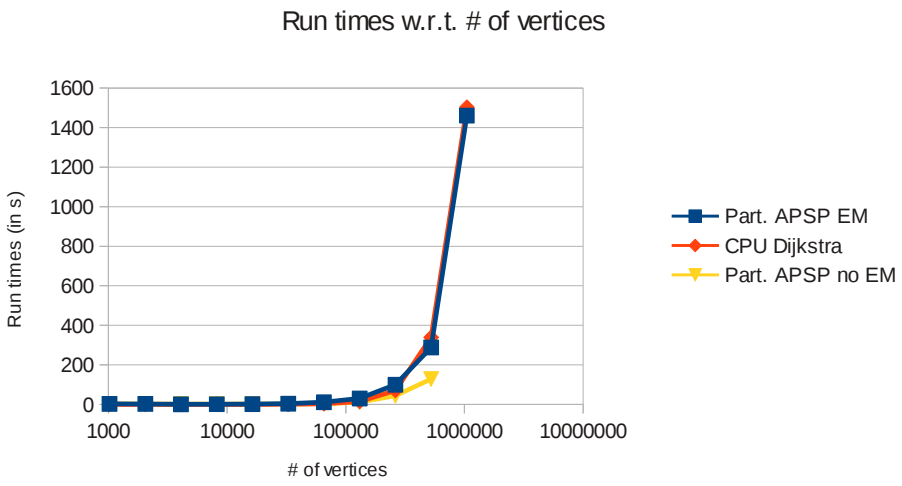

Figure 9: Evolution of run times with respect to the number of vertices. Three implementations are compared: our two implementations - with and without using external memory - and a distributed Dijkstra implementation referred to as CPU Dijkstra. All computations were run on 64 cluster nodes.

In order to test our implementation on a real dataset, we retrieved the Californian road network dataset from [26]. This dataset consists in the entire road network of the state of California; it contains $1,957,027$ vertices corresponding to road intersections and more than 5 million edges corresponding to roads. Computing the 3.8 trillion shortest distances in this network took 31 minutes, using 64 cluster nodes. For this experiment, each node is equipped with two NVIDIA C2090 GPUs, a 16 core Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz and 32 GB of RAM.

5. Decentralized (distributed) approach

5.1. Objectives

A drawback of the centralized (master/slave) approach described in the previous section is that, with a large number of slave nodes, the master node cannot handle the volume of communication and becomes a bottleneck. Specifically, Steps 2, 3 and 4 require that the master node sends sub-matrices to slave nodes

and retrieve temporary results. These requirements induce a large amount of data transfers between the master node and the slave nodes. When a large number of slave nodes are involved, the master node becomes overwhelmed with requests for new tasks, leaving some slave nodes idle. This phenomenon has a negative impact on performances by reducing the scalability of our approach.

Another weakness of the centralized approach is that the distances for the boundary matrix are computed and stored at a single node—the master node. If the memory of the master node is not sufficient to store those distances, of order $\Theta(kn)$, then external memory has to be used, which dramatically reduces the performance.

In order to address those drawbacks, we develop and analyze a decentralized version of the algorithm, where in addition the boundary matrix is computed and stored distributedly.

5.2. Algorithm outline

In our new approach we no longer define a master/slave hierarchy among nodes and instead require that all nodes maintain an updated data structure for the entire graph thereby reducing the communication volume. The new version is described in Algorithm 2. While the overall structure as the same as that of Algorithm 1 and its centralized implementation, there are some important differences, especially in Steps 3 and 4. Also, this version assumes nonneative edge weights. In Step 1, the graph is stored and partitioned at each node independently. Although this does not reduce the computation time, it reduces the communication time as the partitioning information does not need to be send out. Step 2 is essentially the same as Step 2 in Algorithm 2, except that the distributed aspects are explicitly discussed.

Step 3, while computing the same information, i.e., APSP(BG), as in the previous version, does it in a different way. The boundary graph is extracted from information collected in Step 2 independently in each processor (lines 15–17) avoiding the need for communication. Then, at lines 18–22, APSP information is computed in parallel at all nodes N by solving a SSSP problem with source

each vertex assigned to N. While this version of Step 3 offers a comparable level of parallelism as the corresponding step of Algorithm 2, the advantage is that the *whole* distance matrix for BG does not reside at one node, but is rather distributed among all nodes. This is the main reason the new version is more memory efficient, and hence allowing larger graphs to be processed without resorting to external memory, than the old one. Lines 23–26 update the shortest path information for each component to the correct one using the computed distances for BG. This is the same computation as in Algorithm 2, except that there is no need for communication (all data necessary for the computations is already at the nodes where it is needed).

Step 4 does the same computation as in the previous version, except that the amount of data that needs to be transferred is much smaller and it is exchanged directly between the corresponding pairs of nodes, without the need to go through a master node.

5.3. Communication cost analysis

With this approach, some temporary results still need to be exchanged between nodes. Step 2 requires $2k$ transfers of diagonal matrices between the master node and the slave nodes in the centralized implementation, where k is the number of partitions in the original graph, which includes k transfers to send diagonal matrices to slave nodes and another k transfers to retrieve the results. This second implementation reduces the number to k transfers of diagonal matrices for each worker node. We also reduce the size of each transfer, as instead of transferring the entire diagonal matrix, each node will transfer the distances between boundary vertices of the diagonal matrix. Other distances are not required at this point; only the distances between boundary vertices are necessary to build the boundary graph for step 3. At the end of the diagonal computations, the distances between boundary vertices contained in the computed diagonal matrix is broadcast by the node, to which the task was statically attributed, to all remaining nodes. Figure 10 describes the data transfers in step 2 for both implementations. At the end of the computations for the boundary

Algorithm 2 Distributed partitioned All-Pairs Shortest Path algorithm

```
INPUT: A planar graph G(V,E), where the weights of E are non-
    negative
OUTPUT: The distances between the vertices in G

// Step 1
Each processor reads G
Partition G into k roughly equal components (deterministically)
//Step 2
Assign each component to a unique processor
Denote by C(p) the set of components assigned to processor p
for each processor p in parallel and each C∈C(p)
  Floyd-Warshall(C) //compute_APSP(C)
  Send computed boundary distances to all other processors
end for
// Step 3  (distributedly compute distances in BG)
for each processor p in parallel
  Graph BG = extract_boundary_graph(G)
end for
for each processor p in parallel and each C∈C(p)
  for each vertex v in C∩BG
    single_source_dijkstra(v,BG)
  end for
end for
for each processor p in parallel and each C∈C(p)
  Floyd-Warshall(C) //compute_APSP(C)
  Send computed distances to all other processors
end for
// Step 4 (compute between-component distances)
for each processor p in parallel and each C1∈C(p)
  for each component C2 of G
    compute_apsp_between_components(C1,C2)
  end for
end for
```

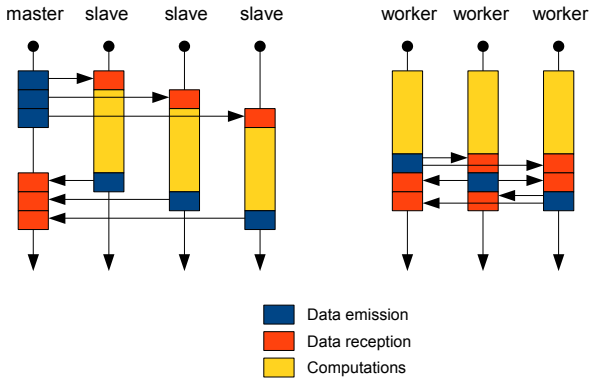

Figure 10: Description of the data transfers in steps 2 and 4 for the centralized implementation (on the left) and the decentralized implementation (on the right).

graph, final distances between vertices from the same component are obtained by computing APSP on the diagonal matrices once more. This time, the entire diagonal matrices are broadcast by the node in charge of the computation for a given diagonal matrix to all other nodes. For step 4, our previous implementation required $2k(k-1)$ data transfers between the master node and slave nodes. This new decentralized implementation does not require any data transfers at this step, since the results for step 4 are final results and do not need to be propagated to other nodes.

The master/slave implementation computes the boundary graph at step 3 using the same partitioned approach recursively. The boundary graph includes virtual edges computed at step 2 of the algorithm, which leads to a higher average vertex degree in the boundary graph than in the initial graph. For each recursive iteration of the algorithm, the density of the boundary graph increases. This increase renders the complexity of the overall algorithm more difficult to estimate. It also requires that the entire boundary graph be allocated on a single node, which for large input graphs can become a bottleneck. In our decentralized approach, we decided to compute the boundary without recursion. Instead, a Dijkstra approach is implemented. Single source Dijkstra computations are executed for each vertex in the boundary graph. Each node is responsible for

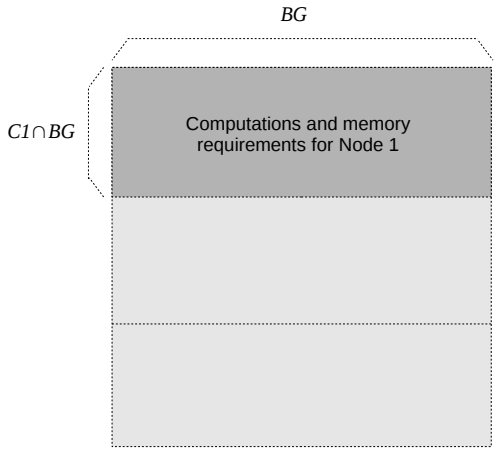

Figure 11: The entire boundary graph distance matrix is distributed among nodes. Each node is in charge of computing and storing the values for a single stripe of the distance matrix corresponding to shortest distances from boundary vertices of a single component to all other boundary vertices. In this example, we have $p = k = 3$.).

the computation of the shortest distances from boundary vertices of components assigned to it to all other boundary vertices. This way memory usage is distributed across nodes as each node only possesses stripes of the entire boundary graph distance matrix - see figure 11.

Single source Dijkstra computations on a single node are executed in parallel across available CPU cores. There is however no guarantee that computing single source Dijkstra for two given vertices in the boundary graph will take the same amount of computations. For this reason, load balancing between nodes may become an issue at this step. This imbalance is however tolerated as any attempt to balance these computations would require transfers between nodes - see algorithm 2 for more details about our decentralized approach.

5.4. Time analysis

Step 2 consists in performing k independent Floyd-Warshall algorithms over graphs of size n/k. The time corresponding to this phase is hence $O(\frac{k}{p}(\frac{n}{k})^3)$. At the end of this step, the nodes broadcast computed boundary distances of their respective components to other nodes. Since the volume of the data re-

lated to each boundary sub-graph is $\sqrt{n/k} \times \sqrt{n/k} = n/k$, and there are k exchanges, the time associated to this transfer is $O(n)$. We consider a model where a broadcasting time is independent of the number of processors; alternatively, one can add an $O(\log p)$ penalty to the total transfer time, to take into account the tree-based communication approach used in some MPI broadcast implementations. The total time of Step 2 is then

$$O\left(n + \frac{n^3}{pk^2}\right), \text{ for } p \leq k. \tag{14}$$

The first stage of Step 3 consists in solving $\sqrt{nk}$ instances of Dijkstra algorithm over the boundary graph BG. As already mentioned, the number of vertices in the boundary sub-graph for each of the k components equals $\sqrt{n/k}$. Hence, the number of edges added for a component is at most $\sqrt{n/k} \times \sqrt{n/k} = n/k$, which gives us n edges for the entire graph. An instance of Dijkstra's algorithm requires then $n \log \sqrt{kn}$ time to be sequentially solved, and since we use p nodes to solve $\sqrt{nk}$ Dijkstra instances in parallel, the total time for this operation is

$$O\left(\frac{n\sqrt{nk}}{p} \log \sqrt{kn}\right) = O(\frac{n^{3/2}k^{1/2} \log n}{p}). \tag{15}$$

The next operation is to compute APSP for diagonal matrices using the Floyd-Warshall algorithm. This operation is then identical to Step 2 and the corresponding time is given by (14).

The computed values (these are the final shortest path distances) are mutually exchanged between the nodes (i.e. k transfers of matrices of size $\sqrt{n/k} \times n/k$). This gives $O(n^{3/2}/k^{1/2})$ time. From (14) and (15) we obtain that, totally, Step 3 requires

$$O\left(\frac{n^{3/2}}{k^{1/2}} + \frac{n^3}{pk^2} + \frac{n^{3/2}k^{1/2} \log n}{p}\right) \tag{16}$$

time.

The computation time for Step 4 is the same as the computation time for Step 4 in the centralized approach, which is

$$O\left(\frac{n^{5/2}}{pk^{1/2}}\right), \text{ for } p \leq k^2 - k. \tag{17}$$

Note that, unlike (10) in the centralized version analysis, in (17) there is no data transmission time.

Summing up equations 14, 16 and 17 we obtain for the total time of Algorithm 2

$$O\left(n + \frac{n^{3/2}}{k^{1/2}} + \frac{n^3}{pk^2} + \frac{n^{3/2}k^{1/2}\log n}{p} + \frac{n^{5/2}}{pk^{1/2}}\right). \tag{18}$$

As for the case of the centralized version, we will try to optimize (18) with respect to k and p. We make the substitution $k = n^\alpha$ and $p = n^\beta$. Then (18) can be written as

$$O\left(n + n^{3/2-1/2\alpha} + n^{3-\beta-2\alpha} + n^{3/2+1/2\alpha-\beta}\log n + n^{5/2-\beta-1/2\alpha}\right). \tag{19}$$

We need to find α and β that minimize the maximum term of (19). To solve that problem, we temporarily drop the $\log n$ term from (19) so that we can solve the resulting linear program. Then, if the fourth term of (19) dominates the solution, we will add the $\log n$ back to the complexity bound; otherwise (the fourth term is of lower order of magnitude) the solution of the linear program will give the time bound since $\log n = o(n^\epsilon)$ for any $\epsilon > 0$.

Then, we are looking for α and β that minimize

$$\max\{1,\, 3/2 - 1/2\alpha,\, 3 - \beta - 2\alpha,\, 3/2 + 1/2\alpha - \beta,\, 5/2 - \beta - 1/2\alpha\}. \tag{20}$$

Since in our implementation p should be at least $2k$, we have to add an additional requirement $\beta \geq \alpha + \log_n 2$. We are, however, dropping the $\log_n 2$ term (it is between 0.05 and 0.1 for graphs of size 10^3 to 10^6), since we don't want to have functions of n in our optimization problem; so the additional constraint added is $\beta \geq \alpha$. Using a linear programming solver, we find that the time is minimized for $\beta = \alpha$ and the time function is decreasing with β (and α), see Figure 12. We see that, with $\alpha = 1$ and $\beta = 1$, the variable t is also 1, which corresponds to running time $T = O(n)$. However, since we ignored one $\log n$ factor from (19) and the fourth term of (20) is equal to the maximum, we add that $\log n$ factor back to the final result, giving running time $T = O(n \log n)$. This is in contrast to the master-slave version (see Figure 6), where there is no improvement in the time beyond $\beta = 0.68$ and the best running time is $O(n^{1.65})$.

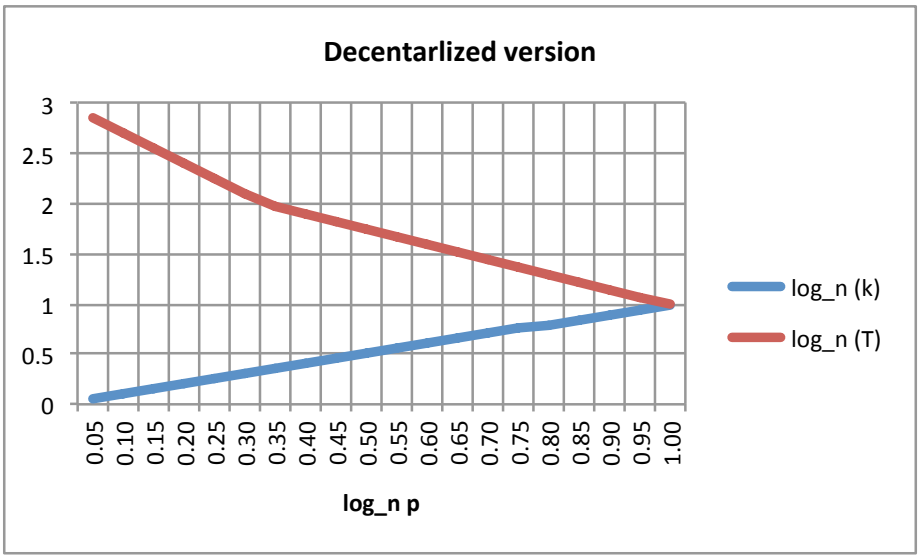

Figure 12: For the decentralized case, optimal values of k and T as functions of the number of processors p, in a log-log scale.

This also shows that our decentralized implementation has the same asymptotic complexity as running Dijkstra's algorithm in parallel using as source vertices all vertices of G. However, as our experiments in Section 4.6 show, our centralized version is much faster than parallel Dijkstra's (see Figure 8). In the following Subsection 5.6 we will show that our decentralized version is even faster than the centralized one.

5.5. Memory analysis

In this approach each node saves $1/k$th of the boundary graph distance matrix (i.e. $\sqrt{n/k}\sqrt{kn} = n$ reals) (see Fig 11) plus k diagonal distance matrices of size $n/k \times n/k$ (i.e. n^2/k reals). The total needed memory is hence

$$O(n + n^2/k). \tag{21}$$

5.6. Results

In order to test our decentralized approach, we design two experiments. Both experiments run on a cluster of more than 300 computer nodes; each node is equipped with two NVIDIA C2090 GPUs, a 16 core Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz and 32 GB of RAM. The first experiments is a comparison with our existing master/ slave approach (using external memory). In this experiment, we increase the size of the graph from 64k to 1024k. The graph size is doubled between two instances and the number of nodes p is multiplied by four - the number of nodes used range from 1 for the 64k vertex instance to 256 for the 1024k vertex instance. Similarly the number of components k is also multiplied by four between two instances and set to $k = 4 * p$. For each instance, the total run time is broken down into run times for every step of the algorithm.

Figure 13 visualizes the results for this first experiment. Even though the master/ slave version was using external memory in this experiment, the last instance did not complete due to memory usage with this number of components. Our new decentralized approach clearly outperforms our master/ slave approach and improves run times by a factor of almost three for the second largest graph instance using the same values for k and the same number of nodes. The largest instance was completed in less than six minutes, ie. four times faster than our fastest run with the master/ slave approach for this instance. These two run times are however more difficult to compare as they were obtained using different values for k and different numbers of nodes.

In a second experiment, we aim at assessing the scalability of our decentralized approach. In this second experiment, the graph size is fixed to 512k vertices and we increase the number of nodes p from 32 to 128. The number of components k is set to $k = 2 * p$; k thus ranges from 64 to 256.

Figure 14 visualizes the results for this second experiment. Run times are again broken down into the run times for the various steps of our approach. The speedups - given in black here and with values taken on the right y-axis - should be interpreted cautiously as the values for k varies between two instances

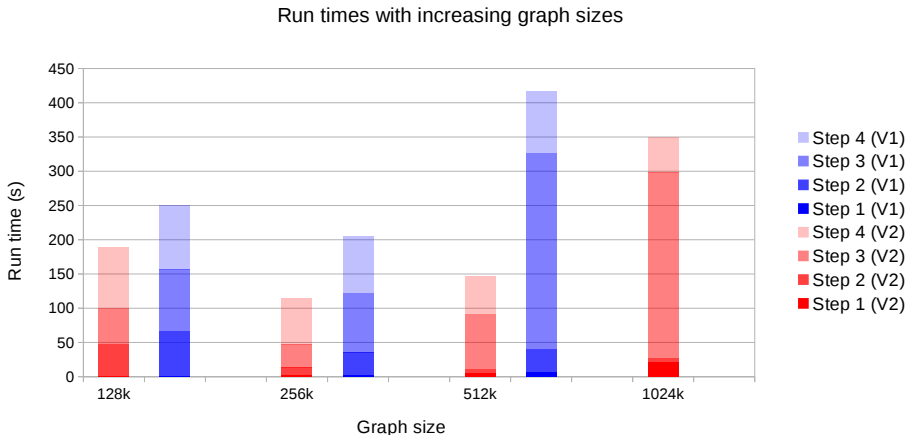

Figure 13: Evolution runtimes with increasing graph sizes. The decentralized version (in red) is compared to the centralized approach (in blue). A breakdown of the run time between the different steps of the algorithm is given.

and has an impact on the overall complexity of the approach. Nevertheless, this decentralized approach clearly benefits from using up to 256 GPUs simultaneously. Though not clearly interpretable, this extended scalability is a definite improvement over our master/ slave approach, which drew almost no benefit from using more than 63 GPUs.

6. Conclusion

We described a new approach for solving the all-pairs shortest path problem on planar and other graphs with good partitioning properties, which is characterized by a nearly optimal number of operations, a regular matrix-structured computations, and which admits a high degree of parallelism. Our implementation on a multi-GPU cluster allows a trillion distances to be computed in half an hour or less, for our centralized version allowing graphs with negative edges weights and under six minutes for our decentralized approach, which oly works with graphs with positive edge weights. Compared with similar algorithms, the proposed here approach is orders of magnitude faster and also allows exploiting a much larger number of GPUs.

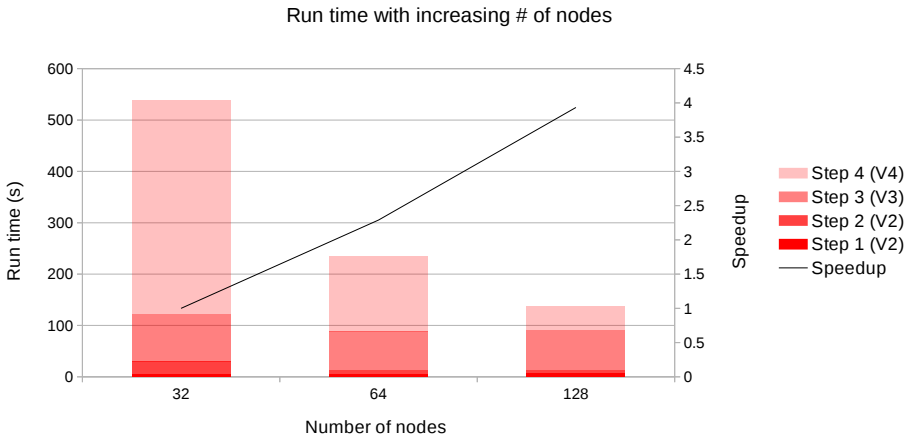

Figure 14: Evolution of run times with increasing numbers of nodes. Speedups (in black) are computing using the run time for 32 nodes as the basis and should be read on the right y-axis.

7. Acknowledgments

We want to thank reviewers for their helpful comments. The authors acknowledge and appreciate the support provided for this work by the Los Alamos National Laboratory Directed Research and Development Program (LDRD). This work was also partially supported by the region of Brittany, France.

References

[1] E. W. Dijkstra, A note on two problems in connexion with graphs, Numerische mathematik 1 (1) (1959) 269–271.

[2] T. H. Cormen, C. Stein, R. L. Rivest, C. E. Leiserson, Introduction to Algorithms, 2nd Edition, McGraw-Hill Higher Education, 2001.

[3] J. Leskovec, D. Huttenlocher, J. Kleinberg, Predicting positive and negative links in online social networks, in: Proceedings of the 19th international conference on World wide web, ACM, 2010, pp. 641–650.

[4] V. Traag, J. Bruggeman, Community detection in networks with positive and negative links, Physical Review E 80 (3) (2009) 036115.

[5] K. Inoue, A. Doncescu, H. Nabeshima, Hypothesizing about causal networks with positive and negative effects by meta-level abduction, in: Inductive Logic Programming, Springer, 2011, pp. 114–129.

[6] D. A. Bader, H. Meyerhenke, P. Sanders, D. Wagner (Eds.), Graph Partitioning and Graph Clustering - 10th DIMACS Implementation Challenge Workshop, Georgia Institute of Technology, Atlanta, GA, USA, February 13-14, 2012. Proceedings, Vol. 588 of Contemporary Mathematics, American Mathematical Society, 2013.

[7] R. J. Lipton, R. E. Tarjan, A Separator Theorem for Planar Graphs, SIAM Journal on Applied Mathematics 36 (2) (1979) 177–189.

[8] P. Harish, P. Narayanan, Accelerating large graph algorithms on the GPU using CUDA, in: High performance computing–HiPC 2007, Springer, 2007, pp. 197–208.

[9] G. J. Katz, J. T. Kider, Jr, All-pairs shortest-paths for large graphs on the gpu, in: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware, GH '08, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 2008, pp. 47–55.

[10] A. Buluç, J. R. Gilbert, C. Budak, Solving path problems on the GPU, Parallel Computing 36 (5) (2010) 241–253.

[11] T. Okuyama, F. Ino, K. Hagihara, A task parallel algorithm for finding all–pairs shortest paths using the GPU, International Journal of High Performance Computing and Networking 7 (2) (2012) 87–98.

[12] K. Matsumoto, N. Nakasato, S. G. Sedukhin, Blocked united algorithm for the all-pairs shortest paths problem on hybrid CPU-GPU systems, IEICE TRANSACTIONS on Information and Systems 95 (12) (2012) 2759–2768.

[13] H. Ortega-Arranz, Y. Torres, D. R. Llanos, A. Gonzalez-Escribano, The all-pair shortest-path problem in shared-memory heterogeneous systems, in:

E. Jeannot, J. Åœilinskas (Eds.), High-Performance Computing on Complex Environments, John Wiley & Sons, Inc., 2014, pp. 283–299.

[14] U. Meyer, P. Sanders, Delta-stepping: a parallelizable shortest path algorithm., J. Algorithms 49 (1) (2003) 114–152.
URL http://dblp.uni-trier.de/db/journals/jal/jal49.html#MeyerS03

[15] K. Madduri, D. A. Bader, J. W. Berry, J. R. Crobak, An experimental study of a parallel shortest path algorithm for solving large-scale graph instances, in: ALENEX, Vol. 7, SIAM, 2007, pp. 23–35.
URL http://dblp.uni-trier.de/db/conf/alenex/alenex2007.html#MadduriBBC07

[16] H. Djidjev, S. Thulasidasan, G. Chapuis, R. Andonov, D. Lavenier, Efficient multi-gpu computation of all-pairs shortest paths, in: 2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014, 2014, pp. 360–369. doi:10.1109/IPDPS.2014.46.
URL http://dx.doi.org/10.1109/IPDPS.2014.46

[17] G. Karypis, V. Kumar, Multilevel k-way partitioning scheme for irregular graphs, Journal of Parallel and Distributed computing 48 (1) (1998) 96–129.

[18] J. L. Gross, T. W. Tucker, Topological Graph Theory, Wiley-Interscience, New York, NY, USA, 1987.

[19] V. Volkov, J. W. Demmel, Benchmarking GPUs to tune dense linear algebra, in: SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing, IEEE Press, Piscataway, NJ, USA, 2008, pp. 1–11. doi:10.1145/1413370.1413402.
URL http://dx.doi.org/10.1145/1413370.1413402

[20] G. N. Frederickson, Fast algorithms for shortest paths in planar graphs, with applications., SIAM J. Comput. 16 (6) (1987) 1004–1022.

[21] M. Snir, S. W. Otto, D. W. Walker, J. Dongarra, S. Huss-Lederman, MPI: the complete reference, MIT press, 1995.

[22] V. Volkov, Better performance at lower occupancy, in: Proceedings of the GPU Technology Conference, GTC, Vol. 10, 2010.

[23] G. L. Miller, Finding small simple cycle separators for 2-connected planar graphs, Journal of Computer and System Sciences 32 (3) (1986) 265–279.

[24] K. Mehlhorn, S. Näher, Leda: A platform for combinatorial and geometric computing, Commun. ACM 38 (1) (1995) 96–102. doi:10.1145/204865.204889.
URL http://doi.acm.org/10.1145/204865.204889

[25] B. Dawes, D. Abrahams, R. Rivera, Boost C++ libraries, URL http://www. boost. org 35 (2009) 36.

[26] J. Leskovec, K. J. Lang, A. Dasgupta, M. W. Mahoney, Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters, Internet Mathematics 6 (1) (2009) 29–123.