Efficient Multi-GPU Algorithm for All-Pairs Shortest Paths

Abstract

I. Introduction

The shortest-path problem is a fundamental computer science problem 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 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 cost 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 of 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 derives from the Floyd-Warshall algorithm, while the second derives from Dijkstra's algorithm. The Floyd-Warshall approach consists in considering going through every vertex v_k of the graph to improve the best known distance between every pair of vertices (v_i, v_j) - see Algorithm 1. The complexity of this approach is $O(|V|^3)$, regardless of the density of the input graph. The algorithm also works for graphs with arbitrary (including negative) edge weights. The cubic complexity of the algorithm however, makes it inapplicable to very large graphs.

The Dijsktra algrithm solves the Single-Source Shortest Paths (SSSP) problem. For a given graph and a given source vertex, it returns the shortest distances from that vertex to all vertices of the graph. Although Dijkstra's algorithm does not solve the APSP problem, it is possible to get the shortest distances for all pairs of vertices by running Dijkstra's algorithm for every vertex of the graph. The Dijkstra algorithm consists in starting from the source vertex and exploring other vertices from closest to farthest incrementally - see Algorithm 2. When using min-priority queues, the complexity of this approach is $O(|E|+|V|log|V|)$ for the SSSP problem. For the APSP problem, the total complexity is thus $O(|V| * |V| + |V|^2 log|V|)$, which becomes $O(|V|^3)$ when the graph is complete. This approach is therefore faster than Floyd-Warshall for sparse graphs.

Algorithm 1 Floyd-Warshall algorithm.

```
INPUT: A 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.

for each vertex v in V
    dist[v][v] = 0
end for
for each edge (u,v) in E
    dist[u][v] = w(u,v)  // the weight of the edge
    (u,v)
end for
for k from 1 to |V|
    for i from 1 to |V|
        for j from 1 to |V|
            dist[i][j] =
     min(dist[i][j], dist[i][k] + dist[k][j])
     end for
  end for
end for
return dist
```

In this paper, we present a new approach to solving the APSP problem for planar graphs that exploits the massive on-chip 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.

Computing the All-Pairs Shortest Path problem is the first step to obtaining many graphs measures that are of importance in many domains such as social network analysis or in bioninformatics with protein-protein interaction networks.

Algorithm 2 Dijkstra's Single Source Shortest Path algorithm.

```
INPUT: A graph G(V,E), where V is a set of
    vertices
  and E a set of weighted edges between these
  vertices. A source vertex from V.
OUTPUT: The distance of the shortest paths between
  the source vertex and every vertex in V.

for each vertex v in V
  dist[v] = infinity
  previous[v] = undefined
end for
dist[source] = 0
Q = V
while Q is not empty
  u = vertex in Q with smallest distance in dist[]
  Q = Q\{u}
  if dist[u] = infinity
    break

  for each neighbor v of u in Q
    alt = dist[u] + dist_between(u, v)
    if alt < dist[v]
      dist[v] = alt
      previous[v] = u
      decrease-key v in Q
    end if
  end for
end while
return dist
```

Furthermore, allowing graphs with negative edge weights to be computed is crucial for many applications. In online social networks, negative edges may be used to indicate antagonism between two individuals [2] or even conflicts and alliances between two groups [3]. Causal networks in bioinformatics also use negative edges to represent inhibitory effects [4].

To harness this computing power for solving path problems in planar graphs, we exploit the community structure of input graphs and reformulate the problem in terms of matrix computations. Our algorithm, based on the Floyd-Warshall algorithm, has near quadratic complexity with respect to the number of nodes, while its 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.

The paper is organized as follows. Section II presents recent parallel implementations for solving the APSP problem. In Section III, we detail the principles of our partitioned algorithm. Section IV focuses on the struture of the data and the computations and how the algorithm is implemented on large multi GPU clusters. Finally, Section V shows the results of two experiments and possible ways to improve our implementation.

II. Related Work

When considering a distributed Graphics Processing Units (GPUs) implementation, both the Floyd-Warshall and Dijkstra 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 requires complex data structures to achieve the best performance. Complex data structures can prove difficult to implement efficiently on a GPU.

Implementing parallel solvers for the APSP problem is an active field of research. [5] 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 the GPUs memory. They report solving APSP for a $100k$ vertex graph in around 22 minutes on a single GPU. [6] 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 [5]. 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. [7] proposed an improvement over the GPU implementation of Dijkstra for APSP from [5] 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[5]. [8] 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 [9] 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.

We propose a novel algorithm and its parallel implementation to compute all shortest distances between all pairs of vertices of a graph with good partitioning properties. The implementation targets executions on large clusters of GPUs in order to handle graphs with up to a million vertices. Experimentations showed 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.

III. Algorithm details

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

A. 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 3). 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 [10]. 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.

B. Step 1: Graph decomposition

In Step 1 the input graph G is divided into k components of roughly equal sizes. The decomposition is done be 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 mportant classes do. These include the class of planar graphs, the class of graphs of low genus, some geometric graphs, and graphs corresponding to networks with good community structure.

Algorithm 3 Partitioned All-Pairs Shortest Path algorithm

```
INPUT: A 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.

function partitioned_APSP(G)
  // Step 1
  for each Component C in G
    compute_APSP(C)
  end for

  // Step 2
  Graph BG = extract_boundary_graph(G)
  compute_apsp(BG)
  for each Component C in G
    compute_APSP(C)
  end for

  // Step 3
  for each Component C1 in G
    for each Component C2 in G
      compute_apsp_between_components(C1, C2)
    end for
  end for
end function
```

C. 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.

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.

D. 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.*

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 1. 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.

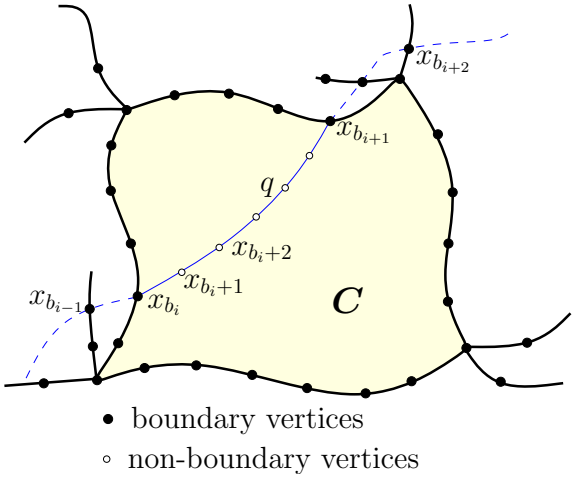

Figure 1. Illustration to the proof of Lemma 1. The shaded region illustrates component C with the subpath $q = (x_{b_i}, x_{b_i+1}, \ldots, x_{b_{i+1}})$ of p inside it.

E. 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 $B_i = C_i \cap B$, $B_j = C_j \cap B$, and*

$$\textit{dist}_4(v_i, v_j) = \min\{\textit{dist}_2(v_i, b_i) + \textit{dist}_3(b_i, b_j) + \textit{dist}_2(b_j, v_j) \mid b_i \in B_i, b_j \in B_j\}. \quad (1)$$

Then $\textit{dist}_4(v_i, v_j)$ 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

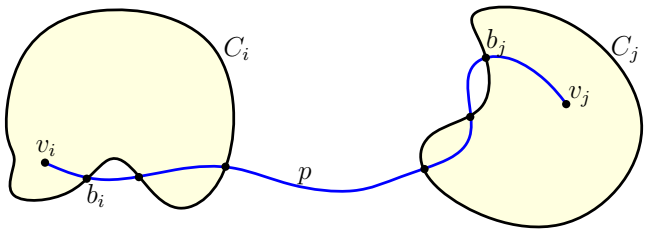

Figure 2. Illustration to the proof of Lemma 2. Note that while in the figure both v_i and v_j are non-boundary, the proof does not make such an assumption.

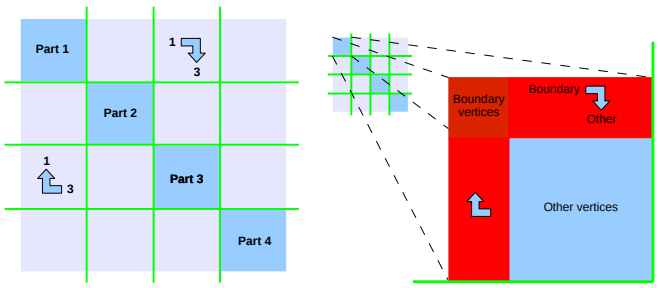

Figure 3. Adjacency matrix after reordering of the vertices. Vertices from the same component are stored contiguously starting with boundary vertices (in red).

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 (eq:step4). Since we don't know which pair (b_i, b_j) of boundary nodes corresponds to the minimum in (eq:step4), 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 $k^2 - k$ 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.

IV. Method

In this section, we first focus on how operations described in the previous section translate in terms of data structure. We then detail the two-level parallel aspect of our implementation. We finally describe the current main memory bottleneck of our approach.

A. Relation to data structure

A simple way to represent a weighted graph is to use an adjacency matrix. For very large graphs however, such a memory intensive representation cannot be considered. 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.

Partitioning the graph is performed using a k-way partitioning routine from the METIS library [10]. Vertices are then reordered so that vertices belonging to a same component are numbered consecutively starting with 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 induces 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 1, 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

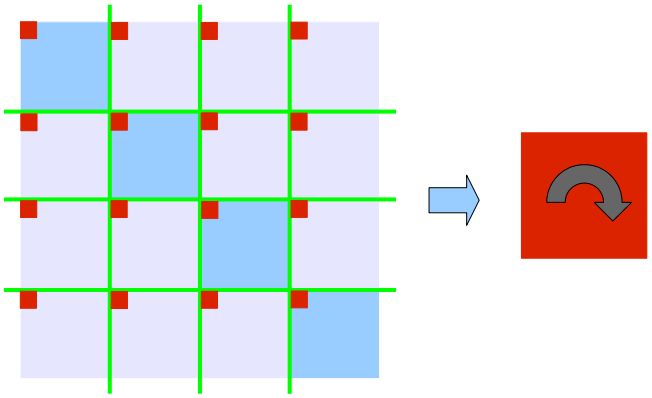

Figure 4. The boundary matrix, here in red, is scattered over the adjacency matrix. Step 2 consits in reconstituting the boundary matrix and computing shortest distances.

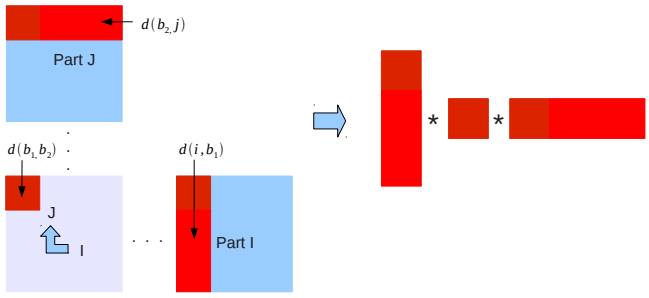

Figure 5. Computations associated to each non-diagonal sub-matrix uses data from 2 diagonal sub-matrices and part of the non-diagonal sub-matrix itself. Computations are similar to matrix multiplications.

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

For step 2, 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 stops when $k = 1$.

For step 3, 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:

- 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.

B. 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-daigonal 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 1 and 3 of our algorithm exhibit interesting parallel properties: a large number of balanced, independent tasks; k tasks for step 1 and $k^2 - k$ for step 3. Using the MPI standard [11], these tasks are distributed accross 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 2, 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 developped by [6] 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 [12] 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, computational power is still available. 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.

C. Memory limitations

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 2, 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 2. 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 3; namely, parts of these sub-matrices containing shortest distances from and to boundary vertices. Shortest distances between non-boundary vertices are thus discarded from main memory at the end of step 2.

The current limiting factor in terms of memory usage is the initial boundary matrix. The first boundary matrix has to fit in main CPU memory. Section V 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.

V. Results and perspectives

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 [13], 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.

Figure 6. 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 [9]. Computations were run using two GPUs on a single cluster node.

Our implementation handles instances up to $512k$ vertices without using external memory. For the very last instance, the use of external memory was 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 [9] 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 6 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 that our implementation is significantly faster than GPU Dijkstra.

Figure 7 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 Part. APSP no I/O implementation. 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

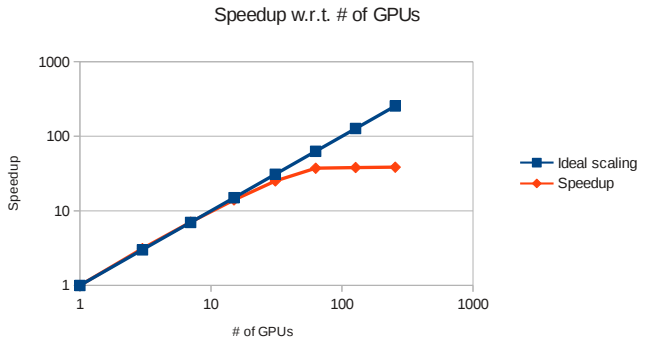

Figure 7. Evolution of speedups with respect to the number of GPUs. The ideal scaling line is given as a reference.

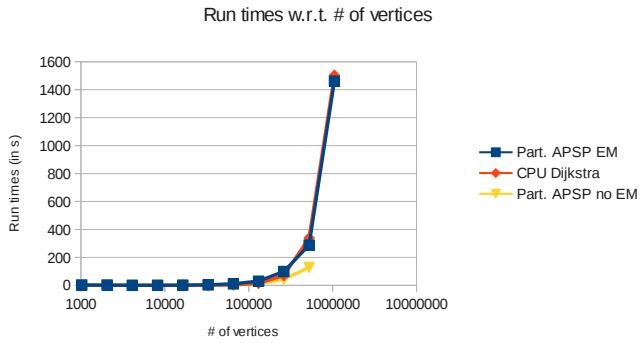

Figure 8. Evolution of run times with respect to the number of vertices. Three implementations are compared: our two implementations - with and without using external memory - and a distributed Dijkstra implementation referred to as CPU Dijkstra. All computations were run on 64 cluster nodes.

parallelism approach that would relieve the master node of some of its communication. A work-stealing approach, for instance, would reduce the amount of communication required for the master node by decentralizing some of the memory transfers. A work-stealing approach is however difficult to implement, due to the two-sided communication scheme enforced by the MPI standard. [14] showed that such an efficient approach was nevertheless feasible. This issue could also 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.

Figure 8 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 [15]. This experiment is not intended to compare directly the performances of 2 GPUs versus a multicore CPU. Instead, we intend to show that our approach is competitive with a distributed Dijkstra approach given a fixed number of heterogeneous cluster nodes. The run times presented in Figure 8 were obtained using 64 cluster nodes. 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 edges to be computed. Our version without external memory is however significantly faster.

REFERENCES

[1] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[2] J. Leskovec, D. Huttenlocher, and 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.

[3] V. Traag and J. Bruggeman, "Community detection in networks with positive and negative links," *Physical Review E*, vol. 80, no. 3, p. 036115, 2009.

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

[5] P. Harish and P. Narayanan, "Accelerating large graph algorithms on the gpu using cuda," in *High performance computing–HiPC 2007*. Springer, 2007, pp. 197–208.

[6] A. Buluç, J. R. Gilbert, and C. Budak, "Solving path problems on the gpu," *Parallel Computing*, vol. 36, no. 5, pp. 241–253, 2010.

[7] T. Okuyama, F. Ino, and K. Hagihara, "A task parallel algorithm for finding all–pairs shortest paths using the gpu," *International Journal of High Performance Computing and Networking*, vol. 7, no. 2, pp. 87–98, 2012.

[8] K. Matsumoto, N. Nakasato, and S. G. Sedukhin, "Blocked united algorithm for the all-pairs shortest paths problem on hybrid cpu-gpu systems," *IEICE TRANSACTIONS on Information and Systems*, vol. 95, no. 12, pp. 2759–2768, 2012.

[9] H. Ortega-Arranz, Y. Torres, D. R. Llanos, and A. Gonzalez-Escribano, "The all-pair shortest-path problem in shared-memory heterogeneous systems," 2013.

[10] G. Karypis and V. Kumar, "Multilevel k-way partitioning scheme for irregular graphs," *Journal of Parallel and Distributed computing*, vol. 48, no. 1, pp. 96–129, 1998.

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

[12] V. Volkov, "Better performance at lower occupancy," in *Proceedings of the GPU Technology Conference, GTC*, vol. 10, 2010.

[13] K. Mehlhorn, S. Näher, and C. Uhrig, "Leda: A platform for combinatorial and geometric computing," vol. 38, 1999.

[14] G. P. Pezzi, M. C. Cera, E. Mathias, and N. Maillard, "On-line scheduling of mpi-2 programs with hierarchical work stealing," in *Computer Architecture and High Performance Computing, 2007. SBAC-PAD 2007. 19th International Symposium on*. IEEE, 2007, pp. 247–254.

[15] B. Dawes, D. Abrahams, and R. Rivera, "Boost c++ libraries," *URL http://www. boost. org*, vol. 35, p. 36, 2009.

VI. Acknowledgments

This work was partially supported by the region of Brittany, France.