



A Parallel Algorithm for Answering Shortest Path Queries in Planar Graphs

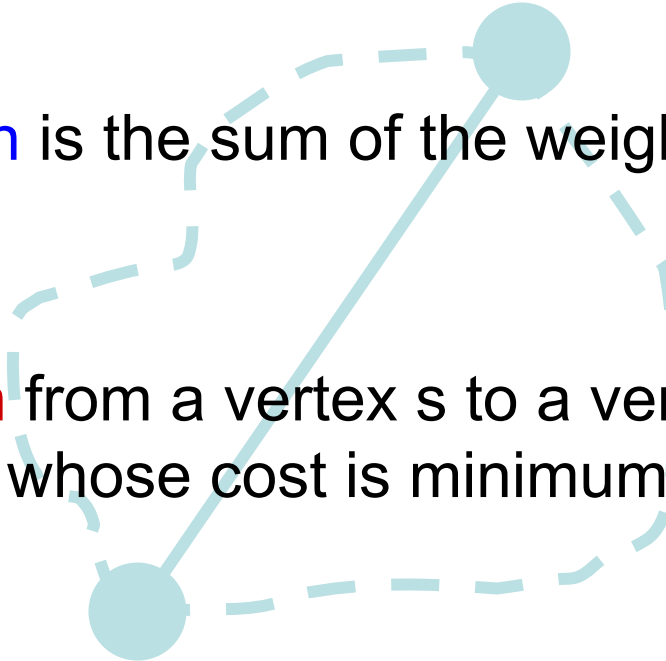
Jon Harris

Contents

- Review & Introduction
- Motivation & Goals
- General Description & Running Times
- Implementation
- Conclusions
- Future Work
- Comments

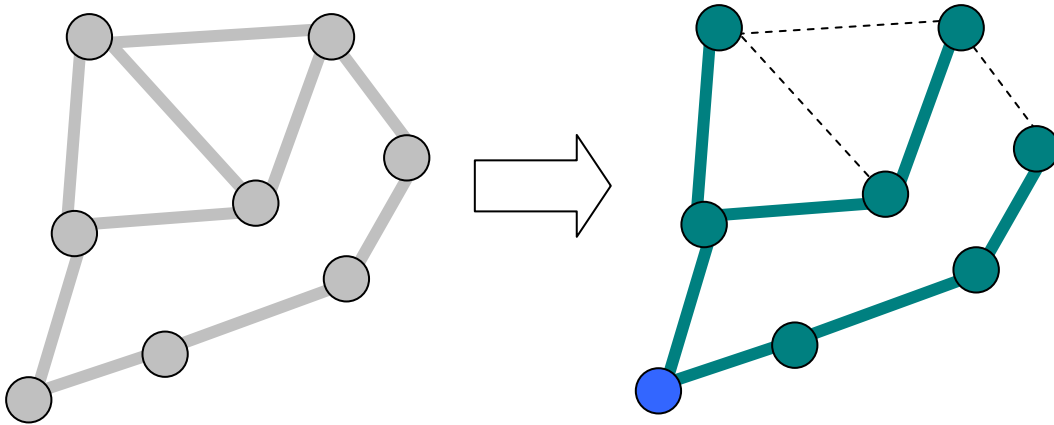
Review

- Let $G=(V,E)$ be a weighted graph with **vertices V** and **edges E** , where each edge has a **non-negative weight**.
- The **cost of a path** is the sum of the weights of the edges along that path.
- The **shortest path** from a vertex s to a vertex t ($t,s \in V$) is a path from s to t whose cost is minimum.



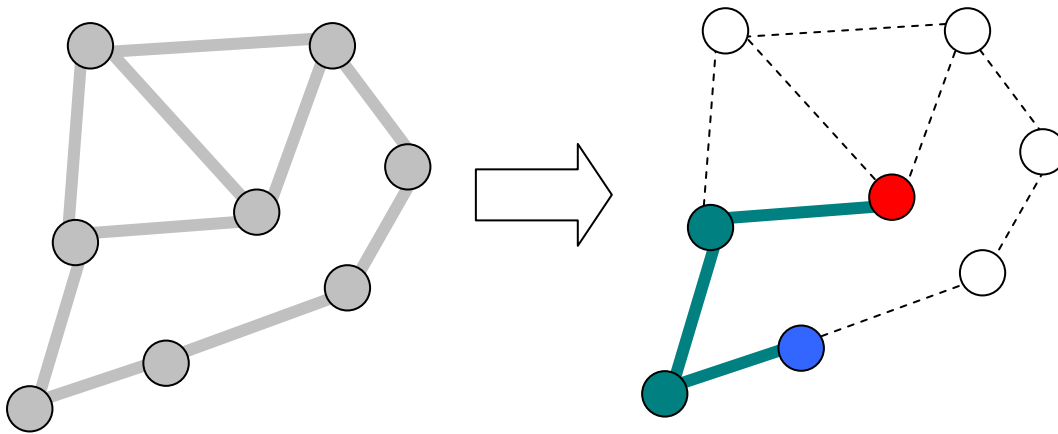
Introduction

- **Single Source shortest path** queries take only a source vertex as input and return the shortest paths from it to all other vertices.



Introduction (cont'd)

- **Simple shortest path queries** take both a source and target vertex as parameters and return the shortest path between those two vertices.



- Dijkstra's single source shortest path algorithm can (sequentially) perform each of the above queries in $O(|V| \text{Log } |V| + |E|)$ time.

Motivation

- Shortest Path problems arise in a variety of contexts.
- Technological advances in physical storage devices have allowed **huge amount of data** to be stored on computers.
- Sequential Shortest Path algorithms are **no longer fast enough** to cope with the huge data sets typical of today's applications.

Goals

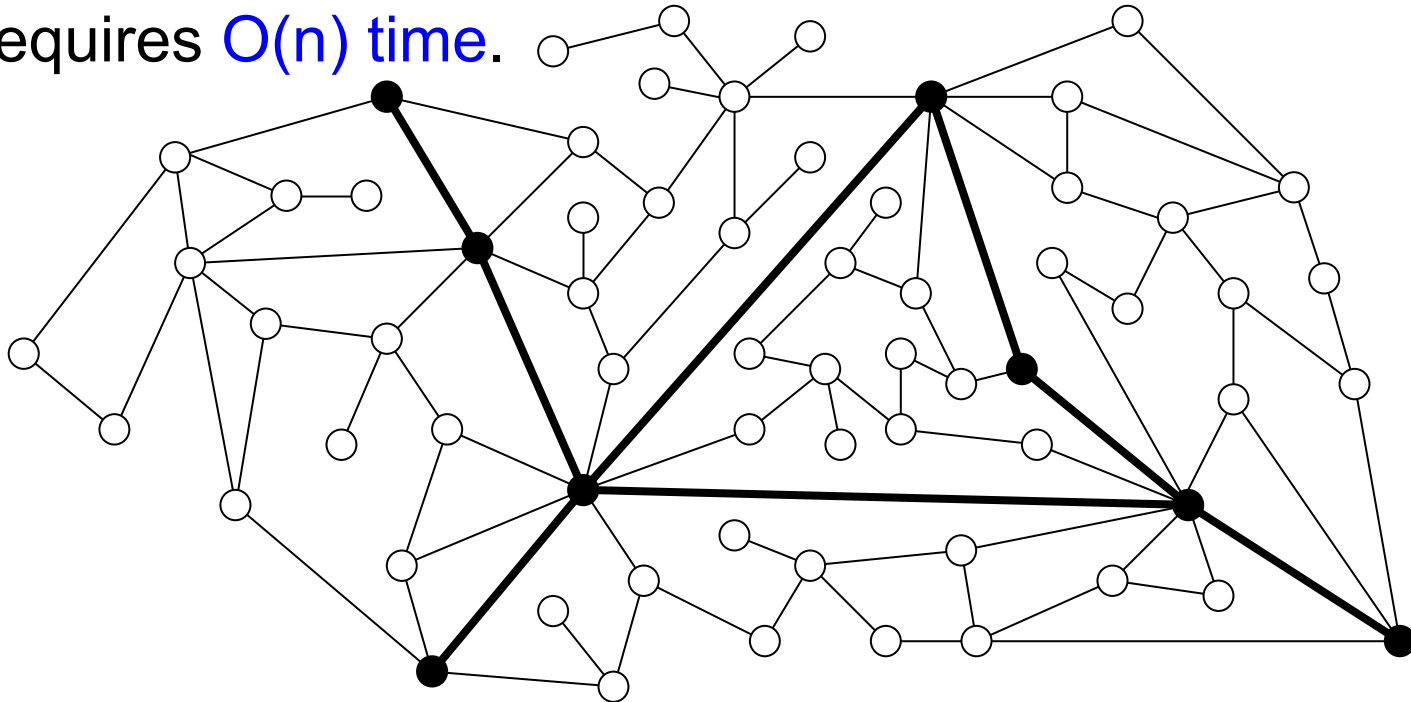
- Improve Shortest Path Query Time Compared to:
 - Sequential Algorithms
 - Existing Parallel Algorithms
- Minimize Storage, Communication Overhead
- Simplify Design Complexity.

Parallel Algorithm

- Proposed by Dr. Lyudmil Aleksandrov
- Designed to run on a **distributed parallel computing model**, adaptation to a **shared memory model** is possible.
- Based on the use of **Graph Separators**
- Consists of several **pre-processing** steps to allow for **fast queries**.
- Repeatedly makes use of Dijkstra's (sequential) single source shortest path algorithm throughout the pre-processing and query stages.

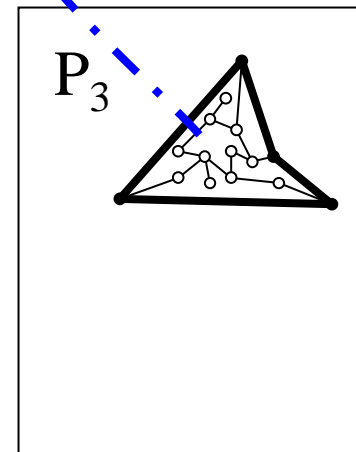
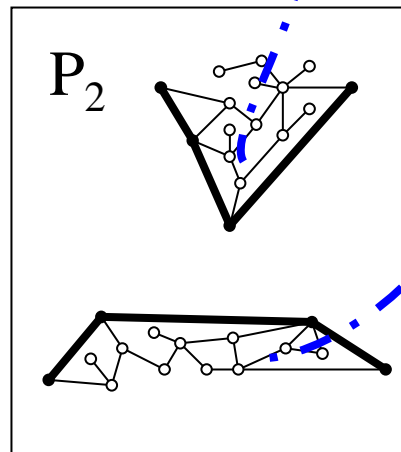
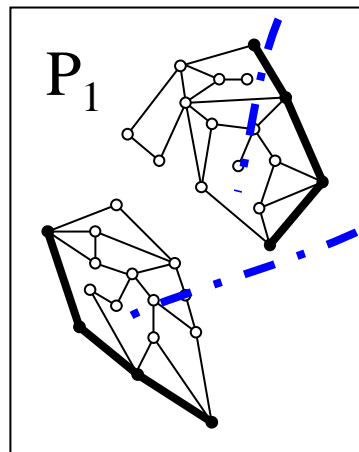
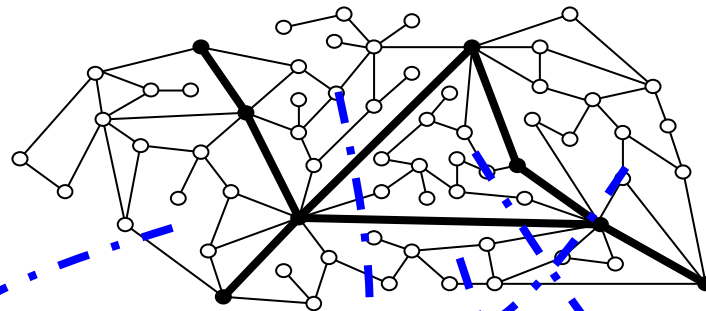
Pre-processing – Step 0

- Partition the input Graph with n vertices into r sub Graphs.
- Each sub Graph has n/r vertices and $(n/r)^{1/2}$ boundary vertices.
- Requires $O(n)$ time.



Pre-processing – Step 1

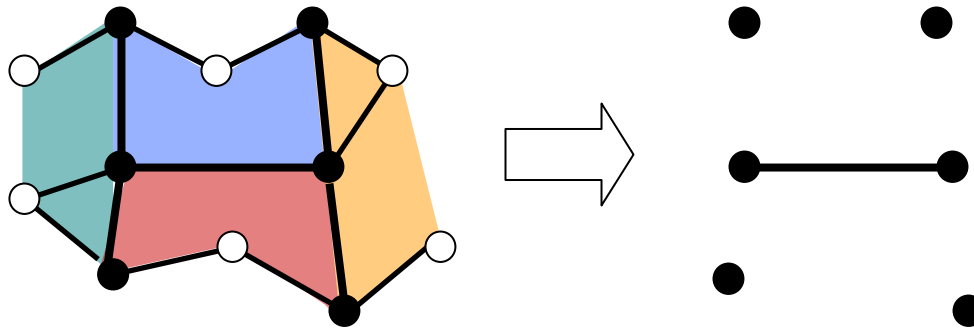
- The partitioned graph is loaded by the master processor.
- The r sub Graphs are assigned to evenly to p worker processors.



$$\begin{aligned} r &= 5 \\ n &= 65 \\ n/r &= 15 \\ (n/r)^{1/2} &= 4 \\ p &= 3 \end{aligned}$$

Pre-processing – Step 1 (cont'd)

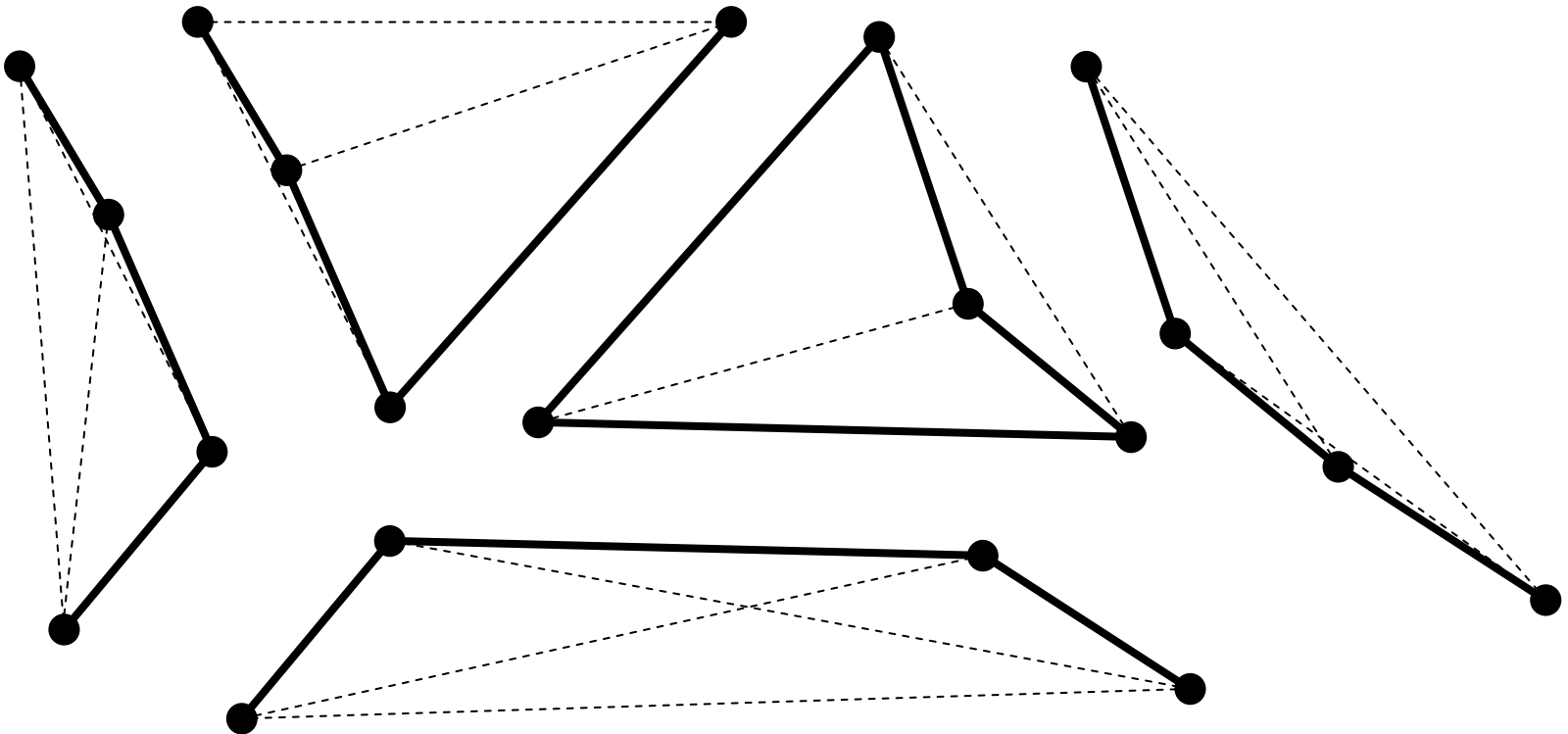
- All boundary vertices of the partitioned graph are transmitted to every worker processor to form the **auxiliary graph G^*** .
- Edges between boundary vertices that are not contained in at least one common sub Graph are also transmitted for G^* .



- Requires **$O(n)$ running time and communication time.**

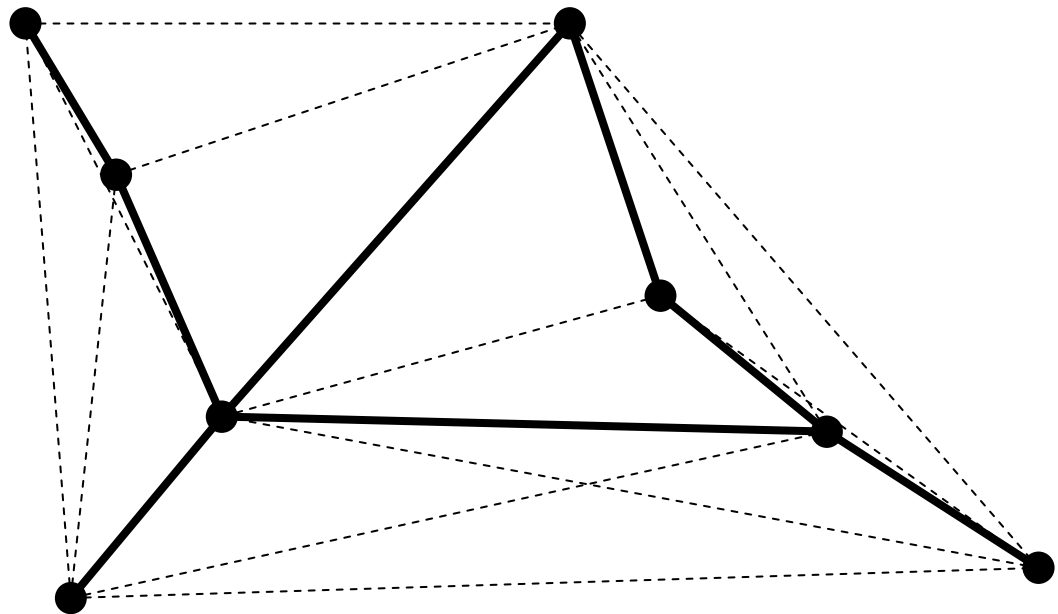
Pre-processing – Step 2

- Each Processor computes the shortest path between all pairs of boundary vertices within each of its sub Graphs.
- Requires $O \left((n / r)^{3/2} \log (n / r) \right)$ running time.



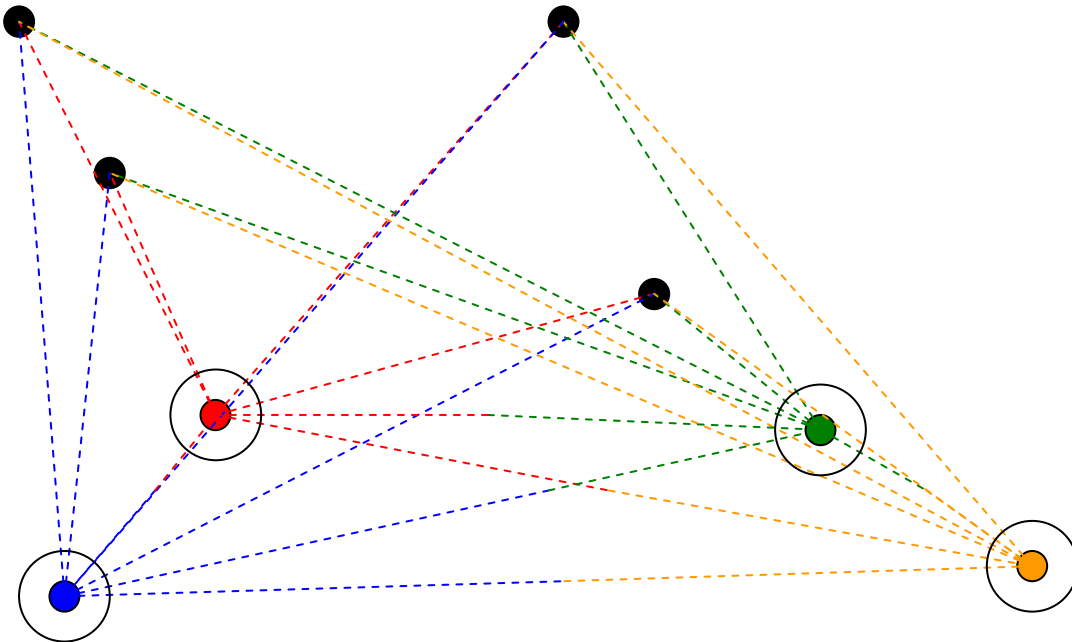
Pre-processing – Step 3

- Each processor exchanges the inter-boundary shortest path information of its sub Graphs with every other processor.
- The exchanged edges are added to the auxiliary graph G^* . When duplicate edges occur, the higher cost edge is discarded.
- Requires $O(n)$ communication time to transmit $O(n)$ edges.



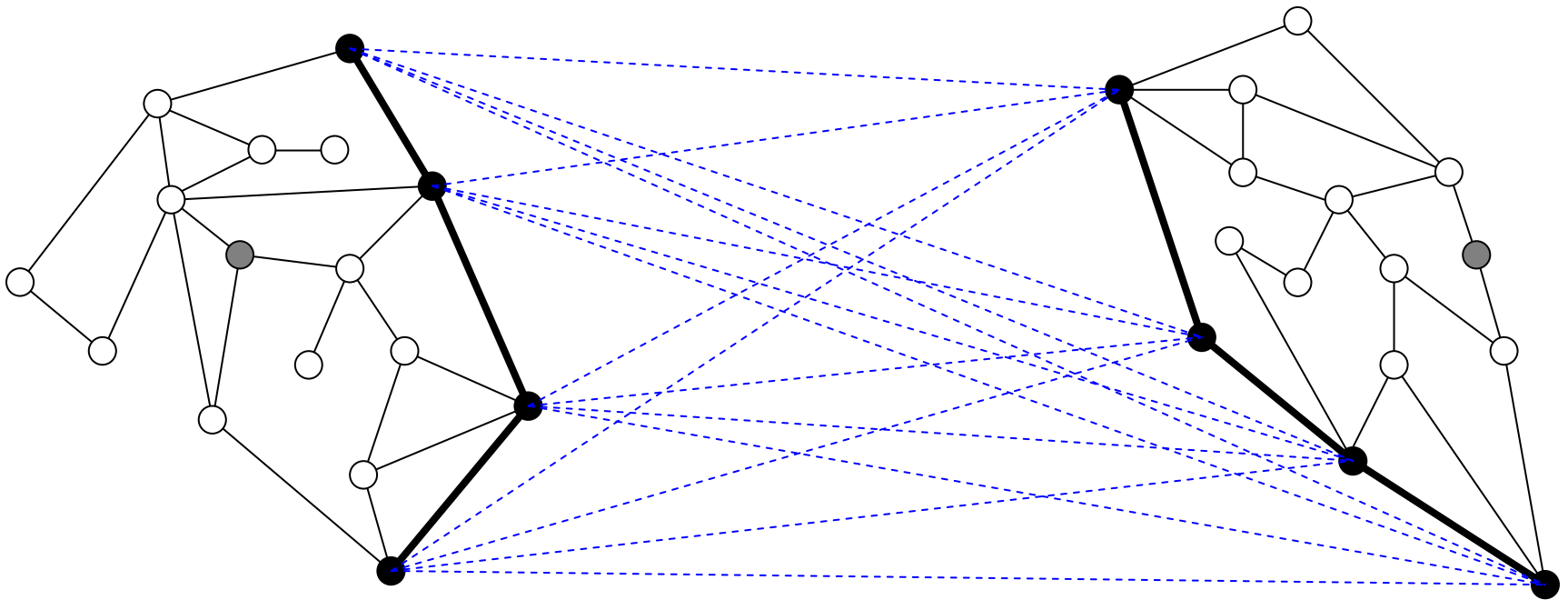
Pre-processing – Step 4

- Each processor computes the shortest path cost from each boundary vertex to every boundary vertex in the auxiliary graph G^* using the edges of G^* .
- Requires $O((n^{3/2} / r^{1/2}) + n \log (nr))$ time.



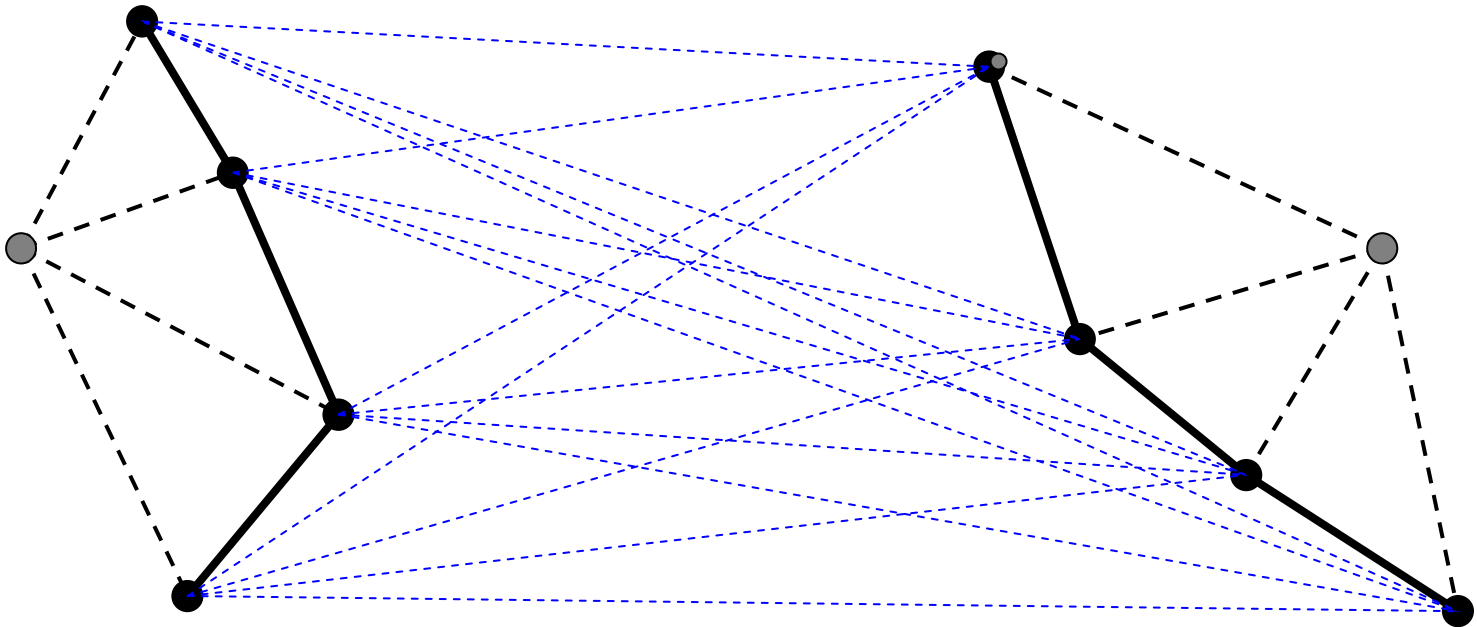
Queries

- Queries are issued to the master processor by providing a **source vertex**, and **optionally** a **target vertex**.
- The master processor forwards the query to the appropriate worker processor(s).



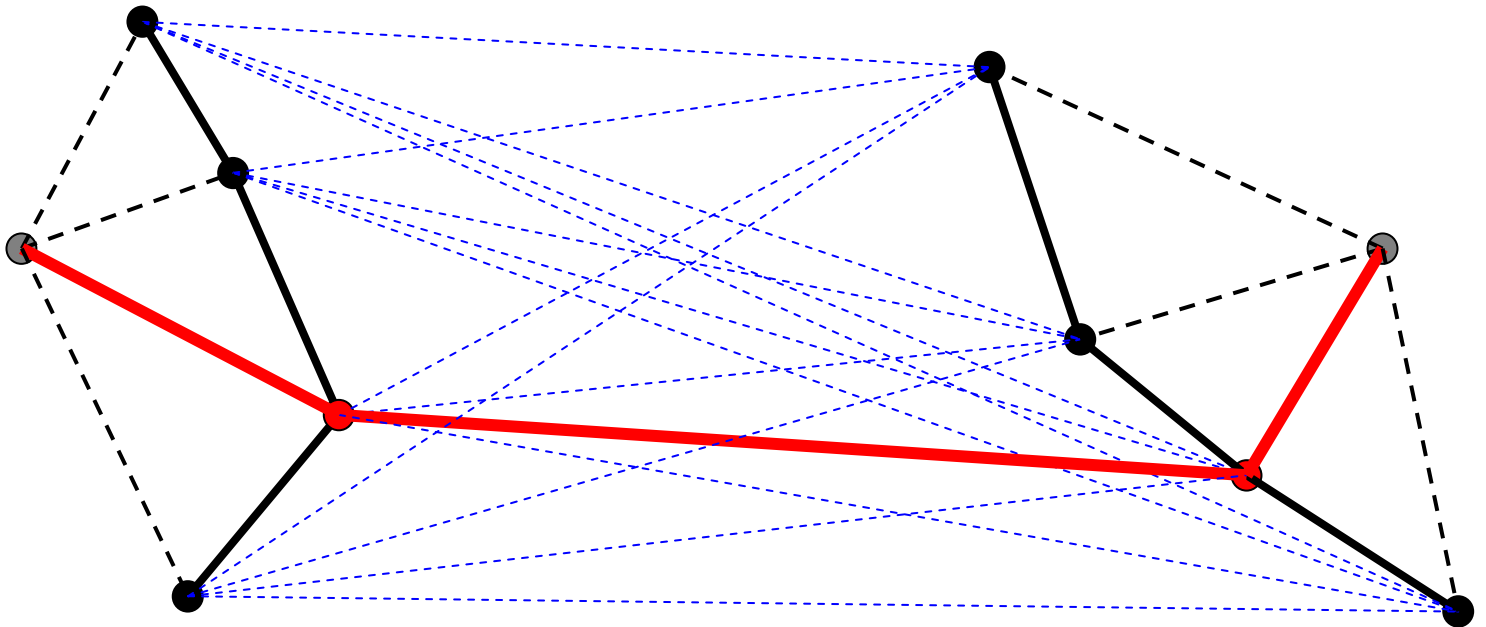
Simple Queries

- The source and target sub Graphs compute the shortest path cost from the query vertex to all of their boundary vertices.
- These costs are then exchanged.



Simple Queries (cont'd)

- The shortest path from the source vertex to a target vertex can now be trivially computed using Dijkstra's sequential shortest path Algorithm.



Simple Queries (cont'd)

- The **running time** of simple queries can be **further improved** by exchanging shortest path costs between the two processors only as they are needed.
- If a request is received for a cost that has not yet been computed, an infinite value can be returned instead.
- One of the two processors will likely compute the shortest path cost to be infinite, but this guarantees that the other processor will correctly compute the shortest path cost afterwards.

Single Source Queries

- Similar to simple queries.
- Source sub Graph computes shortest path costs from source vertex to all boundary vertices.
- This data is transmitted (sorted) to every other sub Graph.
- Each sub Graph continues computing the shortest path from the source sub Graph's boundary vertices to all of its vertices using G^* .

Queries (cont'd)

- Both Simple and Single Source Queries require:
 - $O((n / r) \log (n / r))$ running time.
 - $O((n/r)^{1/2})$ communication time.
- The vertices visited along the paths whose costs are computed can be returned with a little extra work. (storage vs. speed trade-off).

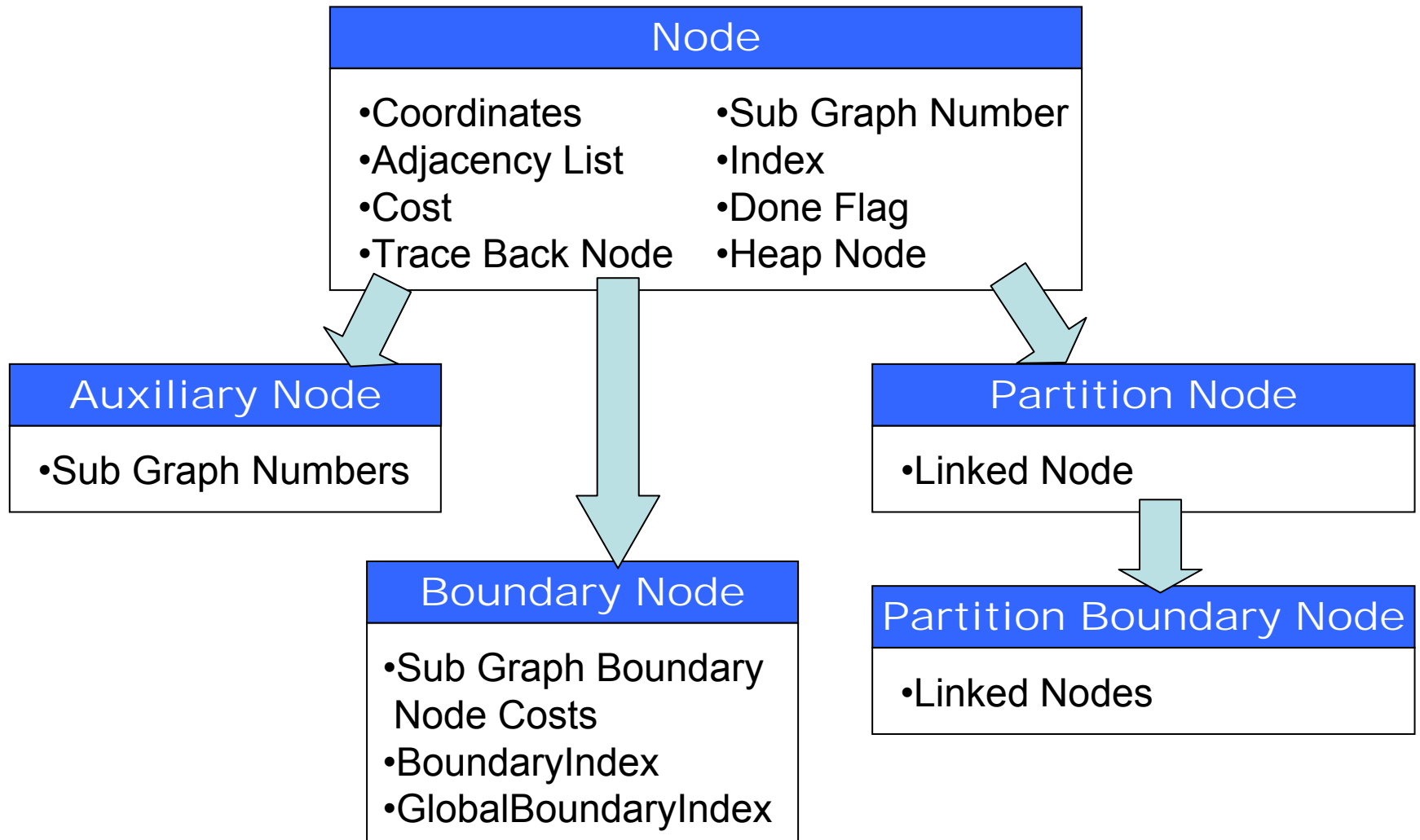
Implementation

- The parallel algorithm was implemented in **C++** using the **MPI** (Message Passing Interface) Parallel library.
- Various 'advanced' features of MPI were used (MPI_AllToAllV, Derived Data Types).
- The implementation was developed for and executed on the **HPCVL Beowulf Cluster** at Carleton University.
- The implementation was also successfully run on the **HPCVL SunFire** (shared memory) Cluster at Carleton University.

Implementation - Data Structures

- [Graph](#) - Stores Node objects.
- [Node](#) - Represents an internal vertex.
- [Boundary Node](#) - Represents a boundary vertex.
- [Aux Node](#) - Represents an auxiliary vertex.
- [Partition Node](#) - An internal vertex from the original Graph, linked to counterpart in a sub Graph.
- [Partition Boundary Node](#) - A boundary vertex from the original Graph, linked to counterparts in sub Graphs.
- [Binary \(Fibonacci\) Heap](#) - Used as the priority queue in the various implementations of Dijkstra's algorithm.

Class Inheritance Hierarchy



Misc. Implementation Details

- Each Auxiliary Node (in G^*) maintains a list of all the sub Graphs it belongs to. This allows the path costs computed step 4 of the pre-processing to be grouped by sub graph.
- Each Boundary Node maintains a Global Boundary Index variable. This allows fast lookup of the Auxiliary Node that corresponds to it.
- Each Partition Node maintains a list of all of the Nodes that correspond to it. This is used for loading the Graph, and determining which sub Graphs query vertices belong to.

Storage Analysis

- Master Processor
 - Original Graph
 - $n - (nr)^{1/2}$ partition nodes
 - $(nr)^{1/2}$ partition boundary nodes
 - Sub Graphs
 - $r \lceil (n/r)^{1/2} \rceil$ boundary nodes
 - n nodes
 - Total
 - $O(n)$ storage (very small constant).

Storage Analysis (cont'd)

- Worker Processor
 - Aux Graph
 - $(nr)^{1/2}$ Auxiliary Nodes
 - n edges between these Nodes
 - Sub Graphs ($\lceil r/p \rceil$)
 - $(n/r) - (n/r)^{1/2}$ Nodes
 - $(n/r)^{1/2}$ Boundary Nodes
 - Total $n + (n/r) - (n/r)^{1/2}$
 - Total
 - $O(n)$ storage per processor (very small constant).

Conclusions & Comments

- Main goals were achieved.
 - Query Time of this algorithm $O((n / r) \log (n / r))$.
 - Query Time of previously fastest known parallel algorithm by Zarliagis and Traff $O(n^{2/3} \log n)$ only on EREW PRAM model.
 - Query Time of Dijkstra's sequential algorithm $O(n \log n)$.
- Implementation was relatively straight-forward.
- Processors behave in a unified manner.
- Communication was kept to a minimum.
- Extremely good load balancing was achieved.

Future Work

- Some / All Pairs Shortest Paths.
- Determine the Vertices occurring along shortest paths whose costs have been computed.