GPU Accelerated Primal-Dual Algorithm

Apoorva Gupta, Jorge Salazar, Jiho Yang

Technische Universität München

Chair of Scientific Computing

October 9, 2017

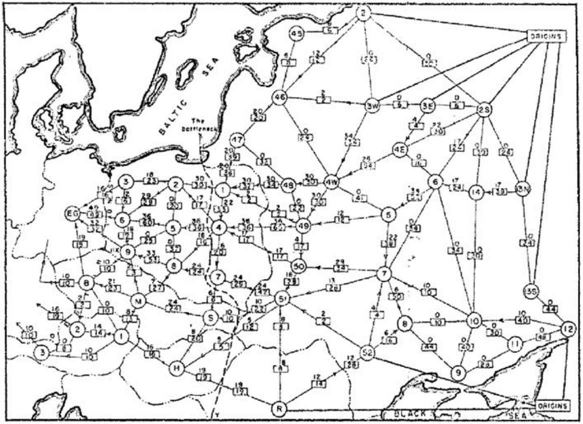
Group members

Group of three CSE students from highly interdisciplinary backgrounds

- Apoorva Gupta - Ambassador of Diplomatic Relations
- Jorge Salazar - Chief Executive Officer
- Jiho Yang - Chief Lavatory & Facilities Manager

Overview of minimum cut maximum flow problem

Soviet Rail Network, 1955



Reference: *On the history of the transportation and maximum flow problems.* Alexander Schrijver in Math Programming, 91: 3, 2002.

Why bother solving minimum cut maximum flow problem?

Minimum cut maximum flow problem gives us a mean to:

- Find the smallest total weight of the edges
- That can, if removed, disconnect the source from the sink

This is done based on the theory that:

- In a flow network, the maximum amount of flow passing from the source to the sink is equal to the total weight of the edges in the minimum cut

Applications of minimum cut maximum flow problem

Innumerous amount of applications including:

- Transportation: minimum number of connections to maximise flow
- Project selection: minimum number of machines to purchase to maximise their usage
- **Image Segmentation**: find pixels assigned to the foreground and background

Mathematical Problem: Maximum flow

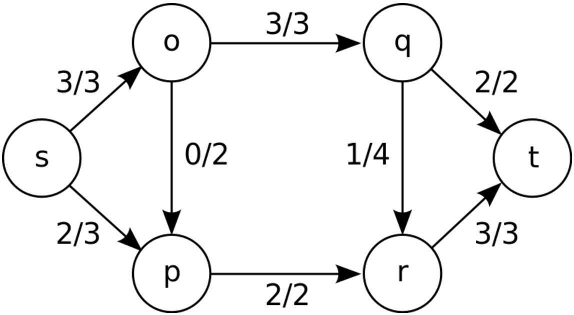
- **INPUT**:
  - Graph $G = (V, E)$, V set of vertices and E set of edges
  - Vertices $s, t \in V$ that denote **source** and **sink** respectively.
  - Capacity function $c : E \to \mathbf{R}^+$

Mathematical Problem: Maximum flow

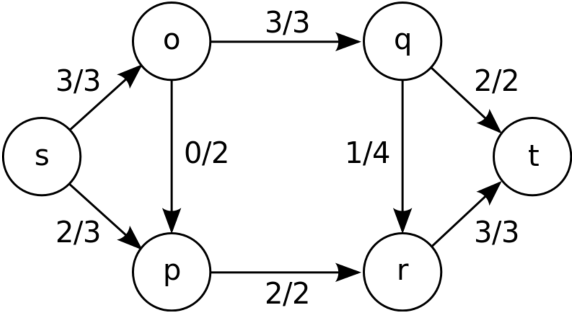
- **INPUT**:
  - Graph $G = (V, E)$, V set of vertices and E set of edges
  - Vertices $s, t \in V$ that denote **source** and **sink** respectively.
  - Capacity function $c : E \rightarrow \mathbf{R}^+$

- **OUTPUT**: Maximum flow $|f|$ from **s** to **t**, such that:
  1. Conservation: $\sum_{(u,v)\in E} f(u,v) = \sum_{(v,u)\in E} f(v,u),\ v \neq s,t$
  2. Capacity: $f(e) \leq c(e),\ e \in E$

Maximum Flow

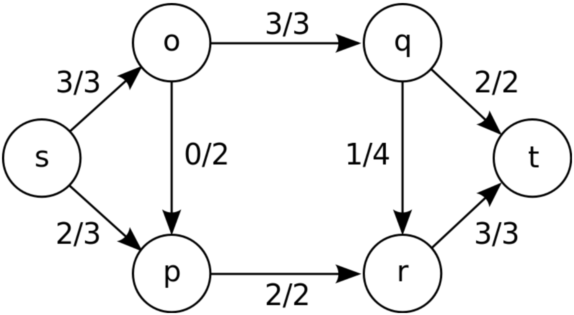


Maximum Flow



- **(1)** "what comes in, comes out"

Maximum Flow



- **(1)** "what comes in, comes out"
- **(2)** the flow cannot exceed the capacity of the edge

Algorithms

It is proven that finding the maximum flow is equivalent to finding the minimum cut between source and sink:

Algorithms

It is proven that finding the maximum flow is equivalent to finding the minimum cut between source and sink:

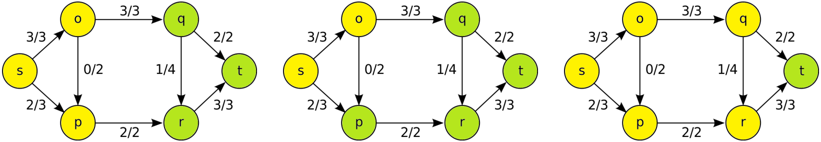


Figure: Different scenarios of cuts

Algorithms

It is proven that finding the maximum flow is equivalent to finding the minimum cut between source and sink:

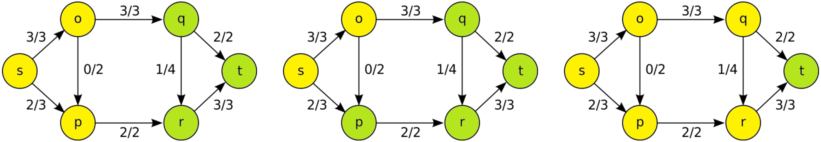


Figure: Different scenarios of cuts

We can reformulate our initial maximization problem to a minimization problem!

Algorithms

We now focus on MinCut algorithms:

1. **IBFS**: Incremental Breadth-First Search
   - State of the art algorithm (2011, 2015), so far the fastest.
   - Based on sequential search in a tree structure to divide the graph and find the minimum cut.
   - Strictly sequential, but provides polynomial run time guarantee.

Algorithms

We now focus on MinCut algorithms:

1. **IBFS**: Incremental Breadth-First Search
   - State of the art algorithm (2011, 2015), so far the fastest.
   - Based on sequential search in a tree structure to divide the graph and find the minimum cut.
   - Strictly sequential, but provides polynomial run time guarantee.
2. **Primal dual**:
   - Simple gradient descent algorithm with preconditioner
   - Highly parallelizable (see next slides)

More about Primal Dual

Make a cut by assigning the label x_i to the node i:

- $x_i = 1$: belongs to the component of the source
- $x_i = 0$: belongs to the component of the sink

Then the minimization function takes the form:

$$E(x) = \sum_{(i,j)\in E_d} \omega(i,j)x_i(1 - x_j)$$

More about Primal Dual

Make a cut by assigning the label x_i to the node i:

- $x_i = 1$: belongs to the component of the source
- $x_i = 0$: belongs to the component of the sink

Then the minimization function takes the form:

$$E(x) = \sum_{(i,j)\in E_d} \omega(i,j) x_i (1 - x_j)$$

With some **magic math**, we can simplify the last equation:

$$E(x) = \langle x, f \rangle + ||\nabla_\omega x||_1 + b$$

where $f \in \mathbb{R}^{|V|}$, $b \in \mathbb{R}^+$ depend only on $\omega(i,j)$.

More about Primal Dual

Given our function to be minimized, now we can compute our gradient descent steps:

More about Primal Dual

Given our function to be minimized, now we can compute our gradient descent steps:

$$
\begin{aligned}
x^{t+1} =& \Pi_{[0,1]^{|V|}}(x^t + diag(\tau_i)(div_\omega y^t - f)) \\
y^{t+1} =& \Pi_{[-1,1]^{|E|}}(y^t + diag(\sigma_{(i,j)})(\nabla_\omega(2x^{t+1} - x^t)))
\end{aligned}
$$

More about Primal Dual

Given our function to be minimized, now we can compute our gradient descent steps:

$$x^{t+1} = \Pi_{[0,1]^{|V|}}(x^t + diag(\tau_i)(div_\omega y^t - f))$$
$$y^{t+1} = \Pi_{[-1,1]^{|E|}}(y^t + diag(\sigma_{(i,j)})(\nabla_\omega(2x^{t+1} - x^t)))$$

We will perform this procedure until we reach a low error or reach a maximum number of iterations.

Gradient and Divergence

- **Gradient:**

$$\nabla_w : \mathbf{R}^{|V|} \to \mathbf{R}^{|E|}$$
$$(\nabla_\omega u)(\{i,j\}) = \omega_{ij}(u_i - u_j)$$

Gradient and Divergence

- **Gradient:**

$$\nabla_w : \mathbf{R}^{|V|} \to \mathbf{R}^{|E|}$$
$$(\nabla_\omega u)(\{i,j\}) = \omega_{ij}(u_i - u_j)$$

- **Divergence:**

$$div_w : \mathbf{R}^{|E|} \to \mathbf{R}^{|V|}$$
$$(div_\omega p)(i) = \sum_{\substack{(i,j)\in E \\ i<j}} \omega_{ij} p_{ij} - \sum_{\substack{(i,j)\in E \\ j<i}} \omega_{ij} p_{ij}$$

Data Structures

Given our graph file, we store ω_{ij}(weights on edges of the undirected graph), neighborhood vertices and adjacent edges of a vertex.

- **Per vertex:** list of adjacent edges and list of nbhd vertices

Data Structures

Given our graph file, we store ω_{ij}(weights on edges of the undirected graph), neighborhood vertices and adjacent edges of a vertex.

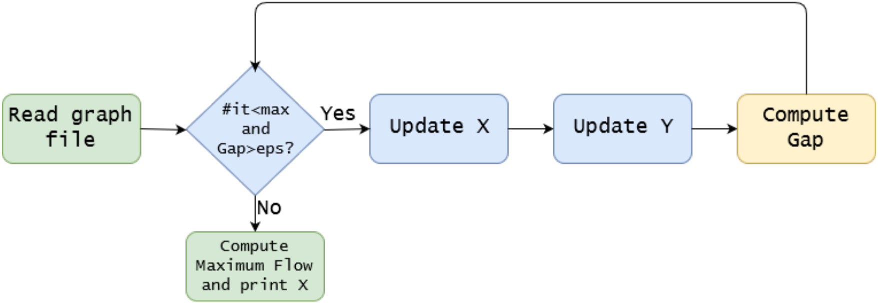
- **Per vertex:** list of adjacent edges and list of nbhd vertices
- **Per edge:** list of start and end vertices

Data Structures

Given our graph file, we store ω_{ij}(weights on edges of the undirected graph), neighborhood vertices and adjacent edges of a vertex.

- **Per vertex:** list of adjacent edges and list of nbhd vertices
- **Per edge:** list of start and end vertices
- ω_{ij}: 1D array of size # edges

Implementation



Implementation: CUDA

- Store ω and f in global memory

Implementation: CUDA

- Store ω and f in global memory
- UpdateX and Compute Gap can de done in parallel by #vertices threads

Implementation: CUDA

- Store ω and f in global memory
- UpdateX and Compute Gap can de done in parallel by #vertices threads
- UpdateY can be done in parallel by #edges threads

Implementation: CUDA

- Store ω and f in global memory
- UpdateX and Compute Gap can de done in parallel by #vertices threads
- UpdateY can be done in parallel by #edges threads
- Global synchronization is necessary between each step of the while loop:

Implementation: CUDA

- Store ω and f in global memory
- UpdateX and Compute Gap can de done in parallel by #vertices threads
- UpdateY can be done in parallel by #edges threads
- Global synchronization is necessary between each step of the while loop: split updateX and updateY into different kernels!

Implementation: CUDA

- Store ω and f in global memory
- UpdateX and Compute Gap can de done in parallel by #vertices threads
- UpdateY can be done in parallel by #edges threads
- Global synchronization is necessary between each step of the while loop: split updateX and updateY into different kernels!
- Use shared/constant memory to store ω_{ij} and f?

Implementation: CUDA

- Store ω and f in global memory
- UpdateX and Compute Gap can de done in parallel by #vertices threads
- UpdateY can be done in parallel by #edges threads
- Global synchronization is necessary between each step of the while loop: split updateX and updateY into different kernels!
- Use shared/constant memory to store ω_{ij} and f? No, lifetime of a block/ too big to be stored

Hardware and Data Specifications

- Graphics card: GeForce GTX Titan X 12GB
- CPU processor: Intel Core i7 2600 3.4 GHz
- Variable type: float

Validation

File Size	Error (%)	t_comp(PD) / t_comp(IBFS)
GB	0.97	2.67
GB	0.46	5.56
GB	0.69	7.78
GB	0.001	4.23
Average	0.53	5.06

Table: Accuracy and Performance of PD-GPU

Results Discussion

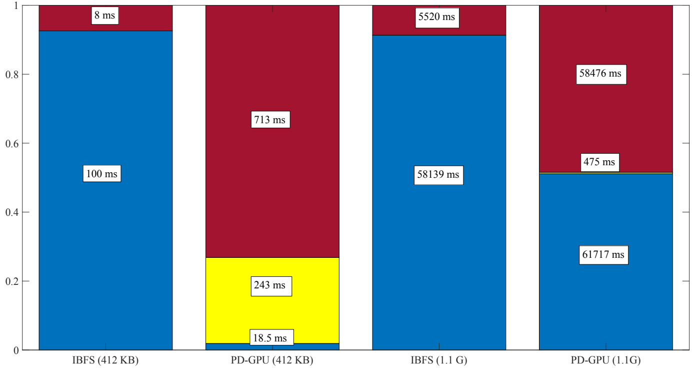


Figure: Normalised instruction time for different problem sizes

Results discussion

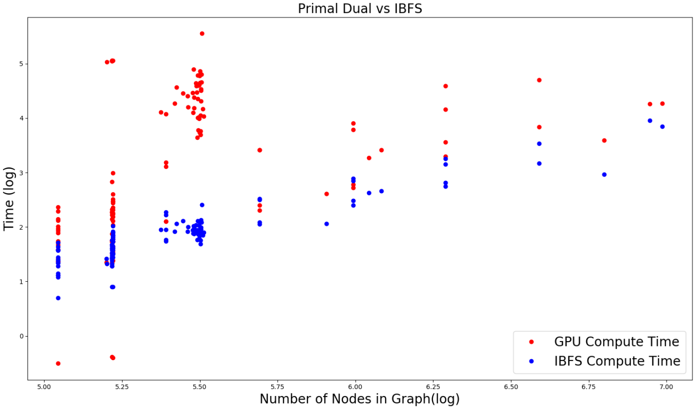


Figure: #nodes vs Time

Results discussion

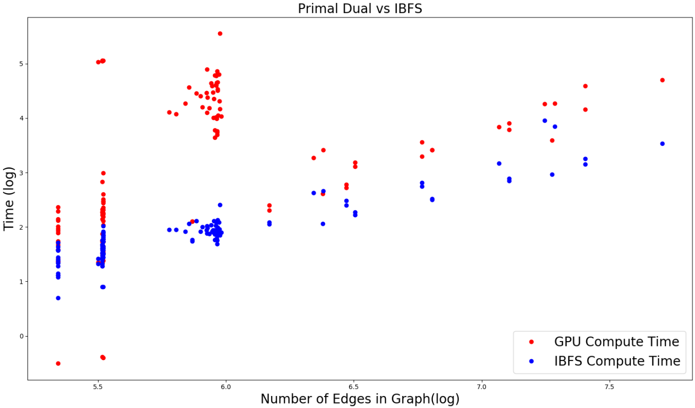


Figure: #edges vs Time

Results discussion

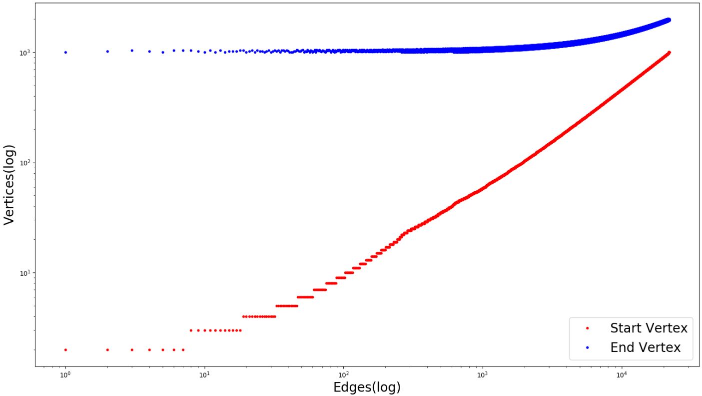


Figure: Edges vs Nodes for file graph3x3.max.bk, PD: 713ms, IBFS: 8ms

Results discussion

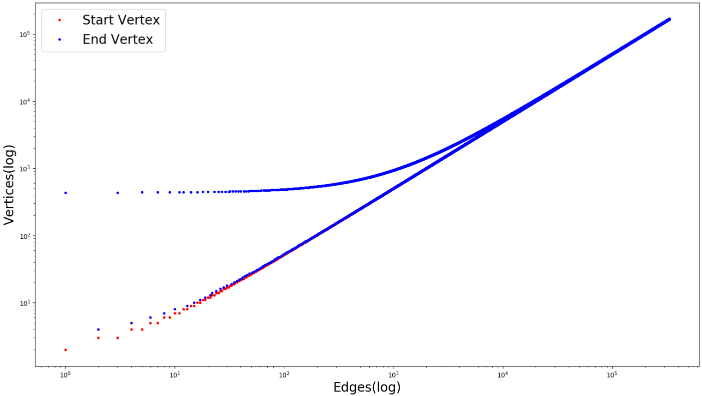


Figure: Edges vs Nodes for file BVZ-venus0.bk, PD:54ms, IBFS:33ms

Results discussion

Figure: Block size effect for file graph3x3.max.bk

Results discussion

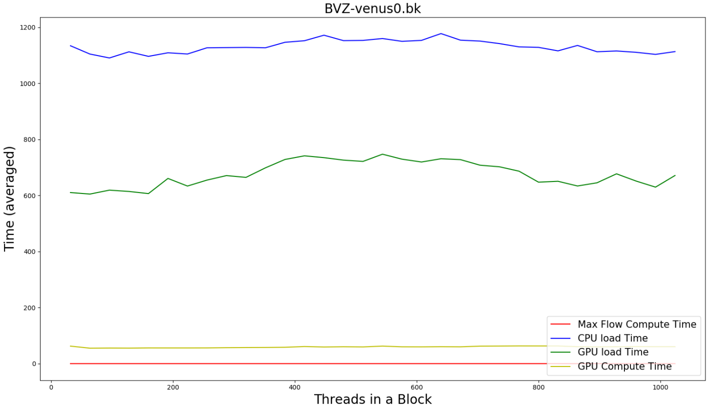


Figure: Block size effect for file BVZ-venus0.bk

Encountered Problems

Several problems were encountered during the project phase:

- Lack of options for code optimization
- Difficulties in understanding and finding the comprehensible source for Primal-Dual Algorithm
- Hyperparameters for Primal-Dual Algorithm - difficult and very heuristic to gain convergence
- Complex data structure
- Lack of C++ library support on CUDA (std::vector, struct, class on device)

Conclusion

- Despite parallel nature of PD it is very difficult to outperform IBFS
- Coalescing a major issue
- BUT coalescing is the only significant optimization available
- Hence no guarantee that coalescing optimization will necessarily make PD outperform IBFS
- Reduction in file load time (inherently sequential) vital
- Anyhow GPU-accelerated PD greatly benefits from multi-threaded computing (bigger the problem size the better)

Future work

Implementation wise:

- Improve coalescing
- Possibility: re-implement to use more cublas functions

Algorithm wise:

- Use Preconditioners
- Basic PD is like using Jacobi Iteration

Thank You. Questions???