A GPU Algorithm for Greedy Graph Matching

B. O. Fagginger Auer R. H. Bisseling

Utrecht University

September 29, 2011

Outline

Introduction

- We will discuss generating greedy graph matchings on the GPU.

Introduction

- We will discuss generating greedy graph matchings on the GPU.
- Graph matching $\approx$ a pairing of neighbouring vertices within a graph.

Introduction

- We will discuss generating greedy graph matchings on the GPU.
- Graph matching $\approx$ a pairing of neighbouring vertices within a graph.
- Matching has applications in

Introduction

- We will discuss generating greedy graph matchings on the GPU.
- Graph matching $\approx$ a pairing of neighbouring vertices within a graph.
- Matching has applications in
  - minimising wireless network power consumption,

Introduction

- We will discuss generating greedy graph matchings on the GPU.
- Graph matching $\approx$ a pairing of neighbouring vertices within a graph.
- Matching has applications in
  - minimising wireless network power consumption,
  - Travelling salesman problem heuristics,

Introduction

- We will discuss generating greedy graph matchings on the GPU.
- Graph matching $\approx$ a pairing of neighbouring vertices within a graph.
- Matching has applications in
  - minimising wireless network power consumption,
  - Travelling salesman problem heuristics,
  - organ donation,

Introduction

- We will discuss generating greedy graph matchings on the GPU.
- Graph matching $\approx$ a pairing of neighbouring vertices within a graph.
- Matching has applications in
    - minimising wireless network power consumption,
    - Travelling salesman problem heuristics,
    - organ donation,
    - ...

Introduction

- We will discuss generating greedy graph matchings on the GPU.
- Graph matching $\approx$ a pairing of neighbouring vertices within a graph.
- Matching has applications in
  - minimising wireless network power consumption,
  - Travelling salesman problem heuristics,
  - organ donation,
  - ...
- Our primary interest is graph coarsening, where we contract matched vertices to obtain a coarser version of the original graph.

Graph Matching

- A graph is a pair $G = (V, E)$ with vertices V and edges E.

Graph Matching

- A graph is a pair $G = (V, E)$ with vertices V and edges E.
- All edges $e \in E$ are of the form $e = \{v, w\}$ for vertices $v, w \in V$.

Graph Matching

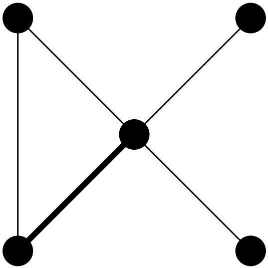
- A graph is a pair $G = (V, E)$ with vertices V and edges E.
- All edges $e \in E$ are of the form $e = \{v, w\}$ for vertices $v, w \in V$.
- A matching is a collection $M \subseteq E$ of edges that are disjoint.

Graph Matching

- A graph is a pair $G = (V, E)$ with vertices V and edges E.
- All edges $e \in E$ are of the form $e = \{v, w\}$ for vertices $v, w \in V$.
- A matching is a collection $M \subseteq E$ of edges that are disjoint.
- We will view matchings as a map $\pi : V \to \mathbb{N}$ such that

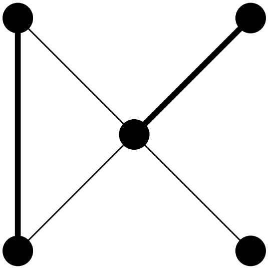
$$\pi(v) = \pi(w) \qquad \Longleftrightarrow \qquad \{v, w\} \in M.$$

Maximal Matching



- A matching is maximal if we cannot enlarge it further by adding another edge to it.

Maximum Matching



- A matching is maximum if it possesses the largest possible number of edges, compared to all other matchings.

Graph Matching

- If the edges are provided with weights $\omega : E \to \mathbb{R}_{>0}$, finding a matching M which maximises

$$\omega(M) = \sum_{e \in M} \omega(e),$$

is called weighted matching.

Graph Matching

- If the edges are provided with weights $\omega : E \to \mathbb{R}_{>0}$, finding a matching M which maximises
$$\omega(M) = \sum_{e \in M} \omega(e),$$
is called weighted matching.
- Greedy matching provides us with maximal matchings, but not necessarily of maximum possible weight or maximum number of vertices/edges.

CPU matching

- We will now look at a serial greedy algorithm which generates a maximal matching.

CPU matching

- We will now look at a serial greedy algorithm which generates a maximal matching.
- In random order, vertices $v \in V$ select and match neighbours one-by-one.

CPU matching

- We will now look at a serial greedy algorithm which generates a maximal matching.
- In random order, vertices $v \in V$ select and match neighbours one-by-one.
- Here, we can pick

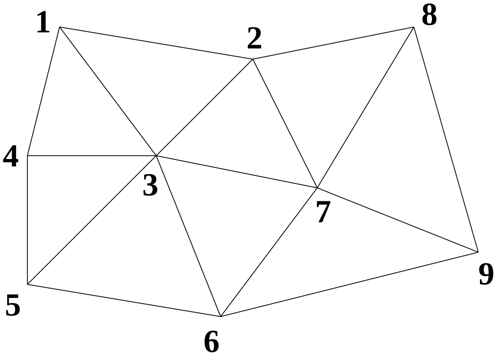
CPU matching

- We will now look at a serial greedy algorithm which generates a maximal matching.
- In random order, vertices $v \in V$ select and match neighbours one-by-one.
- Here, we can pick
  - the first available neighbour w of v (random matching),

CPU matching

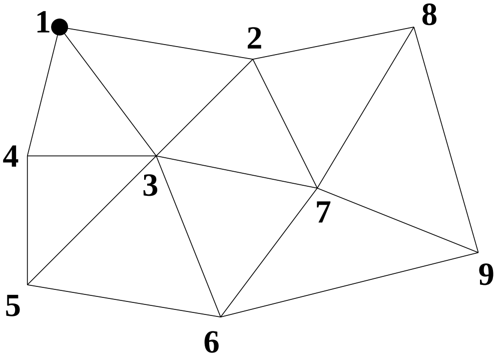
- We will now look at a serial greedy algorithm which generates a maximal matching.
- In random order, vertices $v \in V$ select and match neighbours one-by-one.
- Here, we can pick
  - the first available neighbour w of v (random matching),
  - the neighbour w for which $\omega(\{v, w\})$ is maximal (weighted matching).

CPU matching



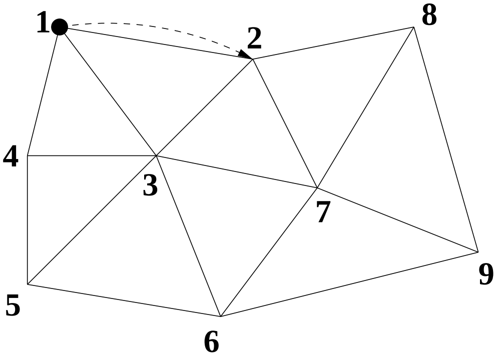
We will create a random matching for this graph.

CPU matching



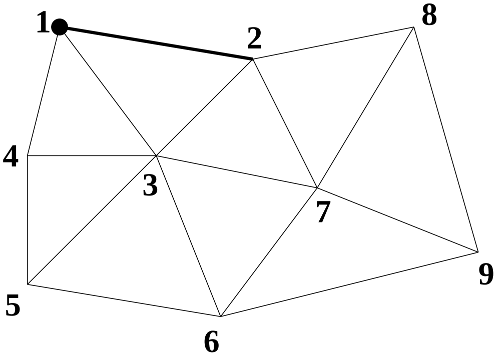
Consider the vertices one-by-one.

CPU matching



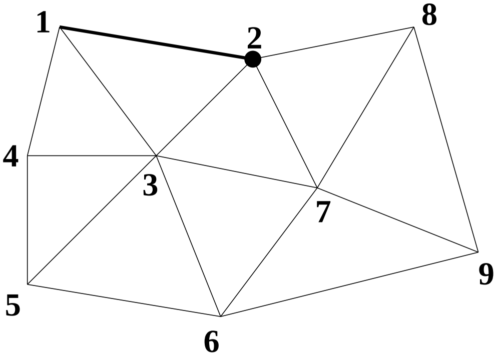
Select unmatched neighbour...

CPU matching



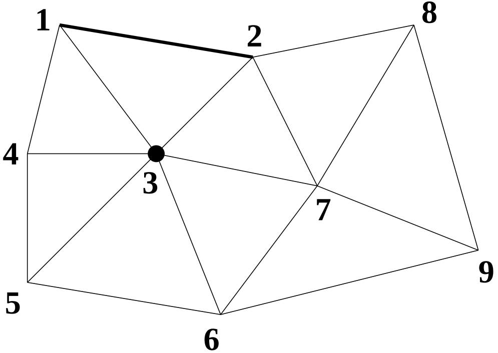
. . . and match.

CPU matching



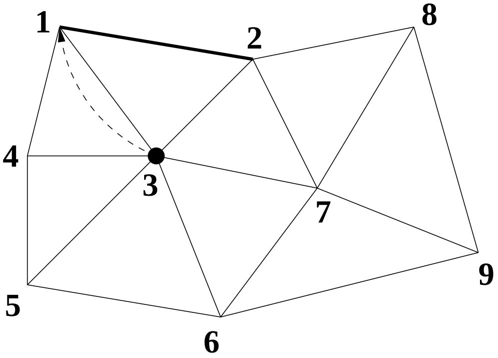
Skip matched vertices.

CPU matching



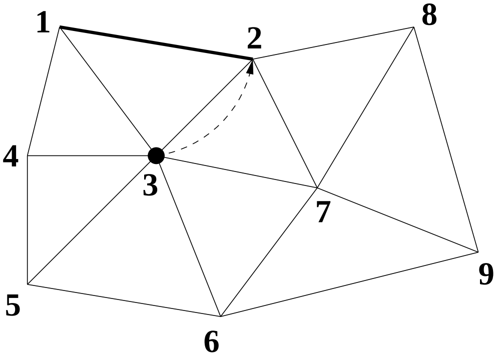
Skip already matched neighbours.

CPU matching



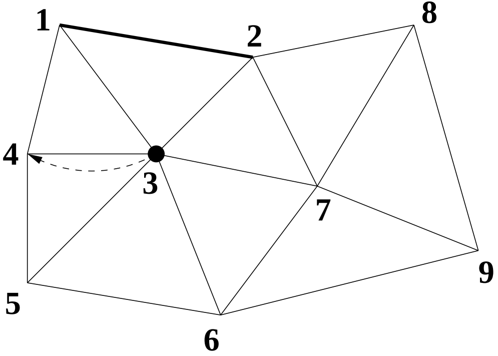
Skip already matched neighbours.

CPU matching



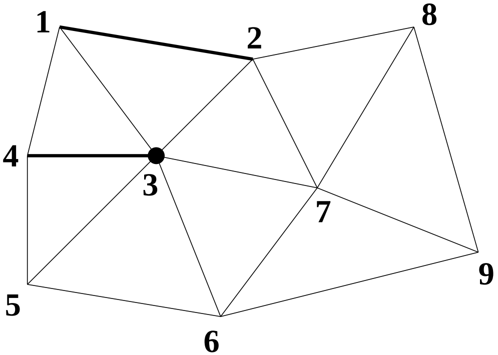
Skip already matched neighbours.

CPU matching



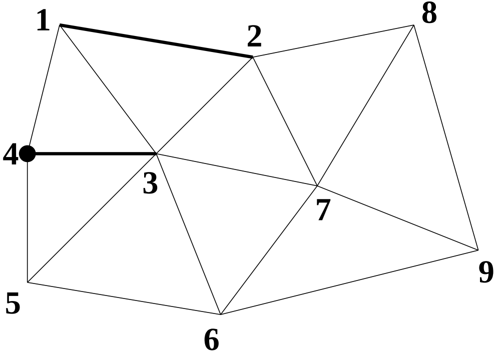
Keep matching until we have treated all vertices.

CPU matching



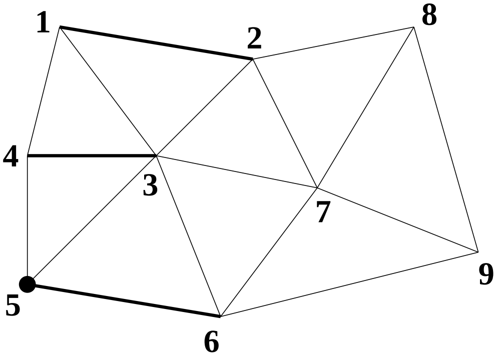
Keep matching until we have treated all vertices.

CPU matching



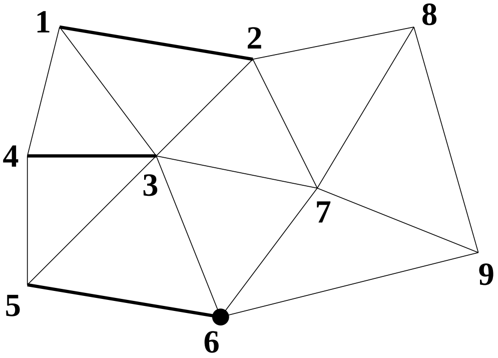
Keep matching until we have treated all vertices.

CPU matching



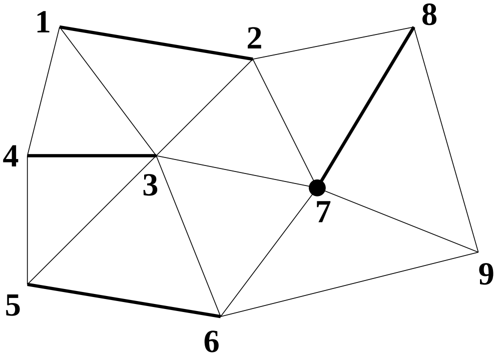
Keep matching until we have treated all vertices.

CPU matching



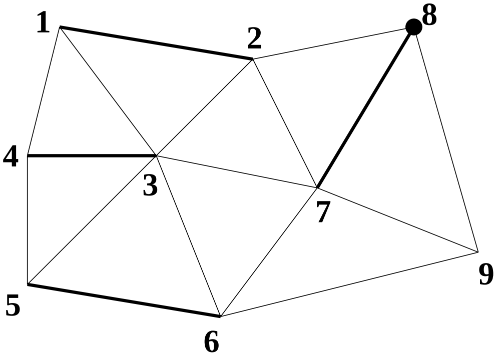
Keep matching until we have treated all vertices.

CPU matching



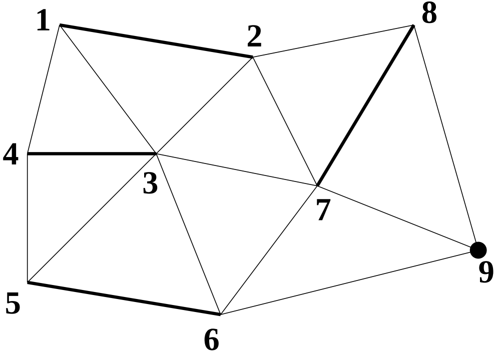
Keep matching until we have treated all vertices.

CPU matching



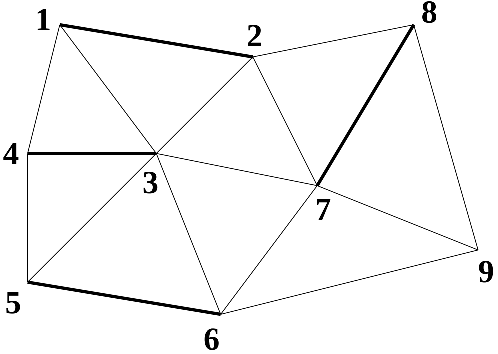
Keep matching until we have treated all vertices.

CPU matching



Keep matching until we have treated all vertices.

CPU matching



We have obtained a maximal matching (also maximum in this case).

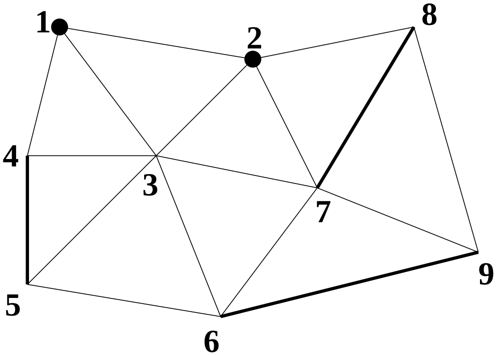
Problematic parallelism

- Directly extending this to a parallel algorithm is problematic.

Problematic parallelism

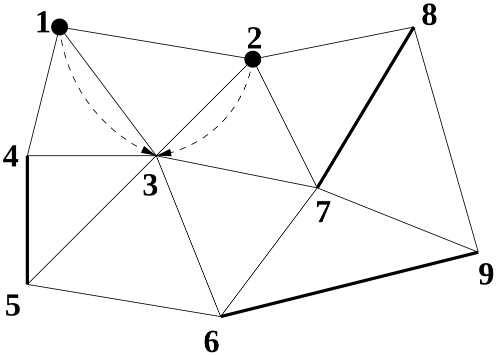
- Directly extending this to a parallel algorithm is problematic.
- Disjoint edges requirement leads to serialisation.

Problematic parallelism



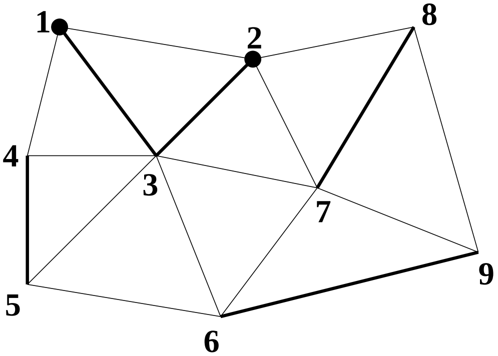
Suppose we match vertices simultaneously.

Problematic parallelism



Vertices find an unmatched neighbour. . .

Problematic parallelism



... but generate an invalid matching.

GPU matching

- To solve this we create two groups of vertices: **blue** and **red**.

GPU matching

- To solve this we create two groups of vertices: **blue** and **red**.
- **Blue** vertices propose.

GPU matching

- To solve this we create two groups of vertices: **blue** and **red**.
- **Blue** vertices propose.
- **Red** vertices respond.

GPU matching

- To solve this we create two groups of vertices: **blue** and **red**.
- **Blue** vertices propose.
- **Red** vertices respond.
- Proposals that were responded to are matched.

GPU implementation

- The graph (neighbour ranges, indices, and weights) is stored as a triplet of 1D textures on the GPU.

GPU implementation

- The graph (neighbour ranges, indices, and weights) is stored as a triplet of 1D textures on the GPU.
- We create one thread for each vertex in V.

GPU implementation

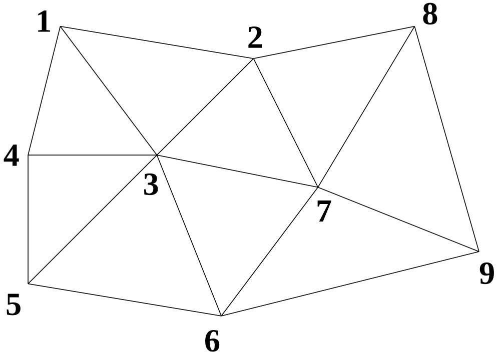
- The graph (neighbour ranges, indices, and weights) is stored as a triplet of 1D textures on the GPU.
- We create one thread for each vertex in V.
- Each vertex $v \in V$ only updates
  - its colour/matching value $\pi(v)$;
  - and its proposal/response value $\sigma(v)$.

GPU implementation

- The graph (neighbour ranges, indices, and weights) is stored as a triplet of 1D textures on the GPU.
- We create one thread for each vertex in V.
- Each vertex $v \in V$ only updates
  - its colour/matching value $\pi(v)$;
  - and its proposal/response value $\sigma(v)$.
- Both π and σ are stored in 1D arrays in global memory.

GPU matching

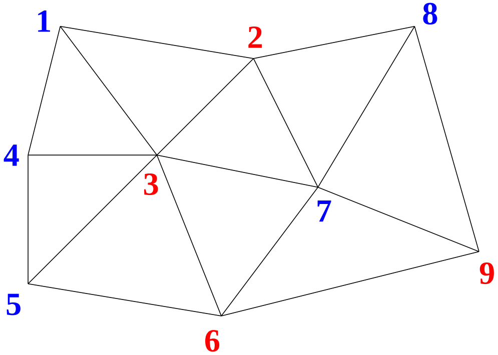
Colour
Propose
Respond
Match



	1	2	3	4	5	6	7	8	9
π	-	-	-	-	-	-	-	-	-
σ	-	-	-	-	-	-	-	-	-

GPU matching

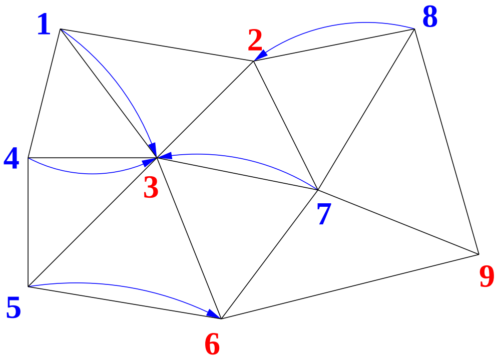
Colour
Propose
Respond
Match



	1	2	3	4	5	6	7	8	9
π	**b**	**r**	**r**	**b**	**b**	**r**	**b**	**b**	**r**
σ	-	-	-	-	-	-	-	-	-

GPU matching

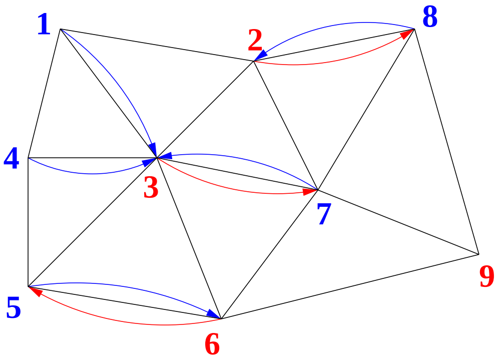
Colour
Propose
Respond
Match



	1	2	3	4	5	6	7	8	9
π	**b**	**r**	**r**	**b**	**b**	**r**	**b**	**b**	**r**
σ	3	-	-	3	6	-	3	2	-

GPU matching

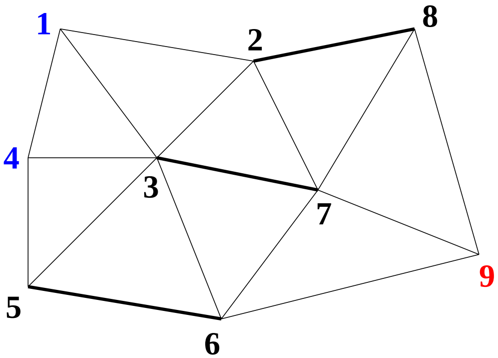
Colour
Propose
Respond
Match



	1	2	3	4	5	6	7	8	9
π	**b**	**r**	**r**	**b**	**b**	**r**	**b**	**b**	**r**
σ	3	8	7	3	6	5	3	2	-

GPU matching

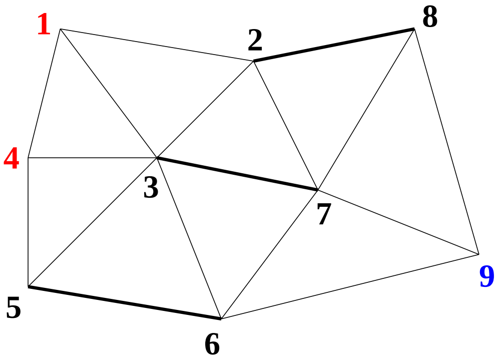
Colour
Propose
Respond
Match



	1	2	3	4	5	6	7	8	9
π	**b**	2	3	**b**	5	5	3	2	**r**
σ	3	8	7	3	6	5	3	2	-

GPU matching

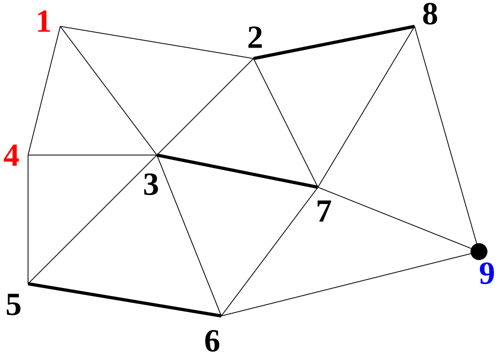
Colour
Propose
Respond
Match



	1	2	3	4	5	6	7	8	9
π	**r**	2	3	**r**	5	5	3	2	**b**
σ	3	8	7	3	6	5	3	2	-

GPU matching

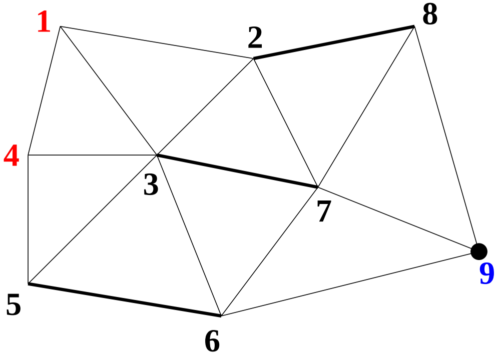
Colour
Propose
Respond
Match



	1	2	3	4	5	6	7	8	9
π	**r**	2	3	**r**	5	5	3	2	**b**
σ	-	-	-	-	-	-	-	-	**d**

GPU matching

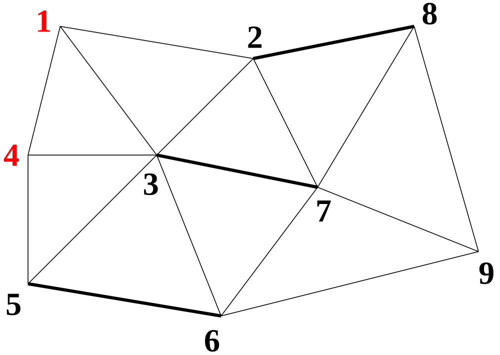
Colour
Propose
Respond
Match



	1	2	3	4	5	6	7	8	9
π	**r**	2	3	**r**	5	5	3	2	**b**
σ	-	-	-	-	-	-	-	-	**d**

GPU matching

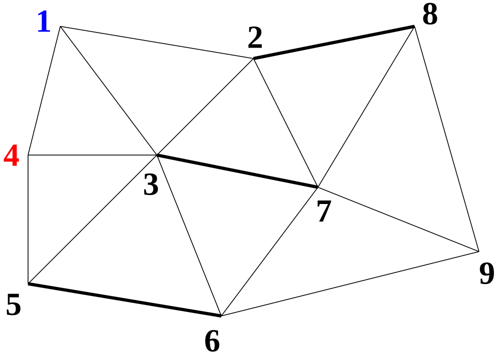
Colour
Propose
Respond
Match



	1	2	3	4	5	6	7	8	9
π	**r**	2	3	**r**	5	5	3	2	**d**
σ	-	-	-	-	-	-	-	-	**d**

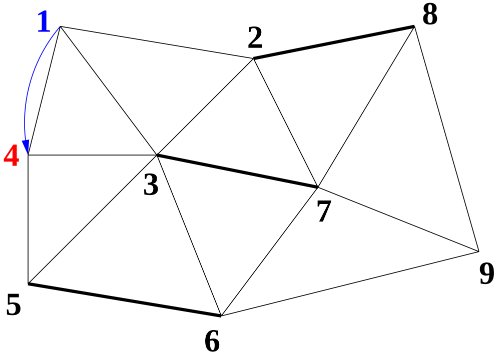
GPU matching

Colour
Propose
Respond
Match



	1	2	3	4	5	6	7	8	9
π	**b**	2	3	**r**	5	5	3	2	**d**
σ	-	-	-	-	-	-	-	-	**d**

GPU matching

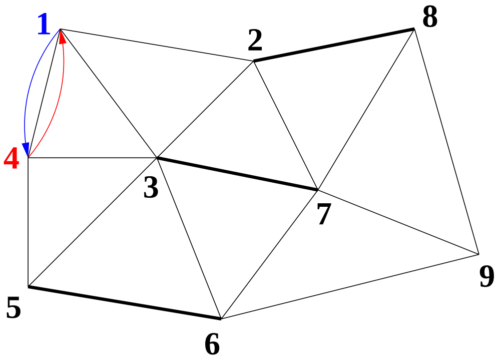


Colour
Propose
Respond
Match

	1	2	3	4	5	6	7	8	9
π	**b**	2	3	**r**	5	5	3	2	**d**
σ	4	-	-	-	-	-	-	-	-

GPU matching

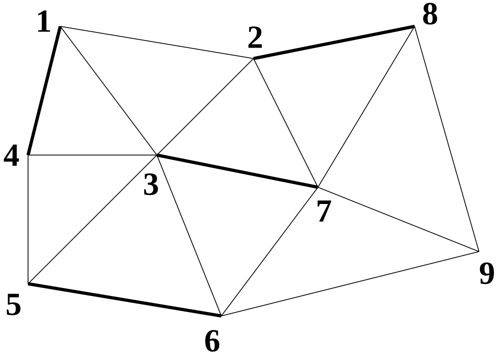
Colour
Propose
Respond
Match



	1	2	3	4	5	6	7	8	9
π	**b**	2	3	**r**	5	5	3	2	**d**
σ	4	-	-	1	-	-	-	-	-

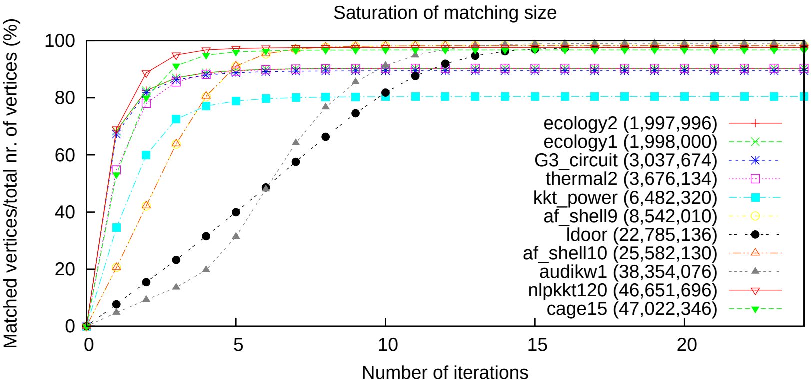
GPU matching

Colour
Propose
Respond
Match



	1	2	3	4	5	6	7	8	9
π	1	2	3	1	5	5	3	2	**d**
σ	4	-	-	1	-	-	-	-	-

Matching saturation



Fraction of matched vertices as function of the number of iterations.

Colouring vertices

- To colour vertices $v \in V$, we use for a fixed $p \in [0, 1]$

$$\textbf{colour}(v) = \begin{cases} \textbf{blue} & \text{with probability } p, \\ \textbf{red} & \text{with probability } 1 - p. \end{cases} \tag{1}$$

Colouring vertices

- To colour vertices $v \in V$, we use for a fixed $p \in [0, 1]$

$$\textbf{colour}(v) = \begin{cases} \textbf{blue} & \text{with probability } p, \\ \textbf{red} & \text{with probability } 1 - p. \end{cases} \tag{1}$$

- How to choose p? Maximise the number of matched vertices.

Colouring vertices

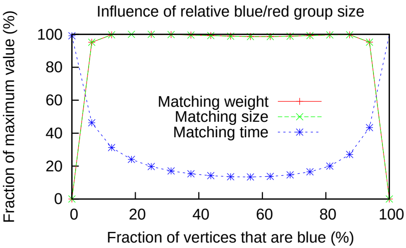
- To colour vertices $v \in V$, we use for a fixed $p \in [0, 1]$

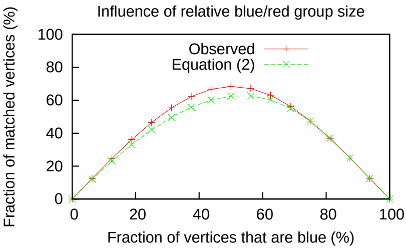
$$\textbf{colour}(v) = \begin{cases} \textbf{blue} & \text{with probability } p, \\ \textbf{red} & \text{with probability } 1 - p. \end{cases} \tag{1}$$

- How to choose p? Maximise the number of matched vertices.
- For a large random graphs, the expected fraction of matched vertices can be approximated by (independent of edge density)

$$2\,(1-p)\left(1 - e^{-\frac{p}{1-p}}\right). \tag{2}$$

Choosing p





Equation (2): we should choose $p \approx 0.53406$.

Results

- Created an implementation on the GPU using CUDA and on the CPU using TBB.

Results

- Created an implementation on the GPU using CUDA and on the CPU using TBB.
- We consider both random and weighted matching.

Results

- Created an implementation on the GPU using CUDA and on the CPU using TBB.
- We consider both random and weighted matching.
- Vertex orderings are randomised and results are averaged over 32 randomisations.

Results

- Created an implementation on the GPU using CUDA and on the CPU using TBB.
- We consider both random and weighted matching.
- Vertex orderings are randomised and results are averaged over 32 randomisations.
- Time only pertains to matching, not I/O or randomisation.

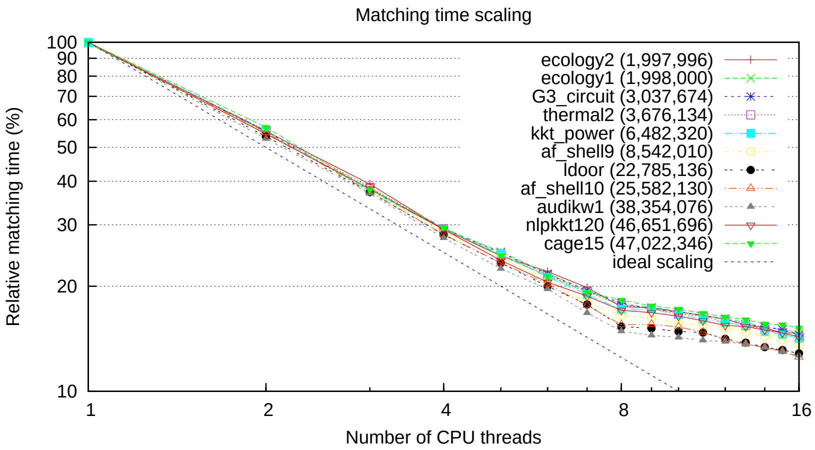
Results

- Created an implementation on the GPU using CUDA and on the CPU using TBB.
- We consider both random and weighted matching.
- Vertex orderings are randomised and results are averaged over 32 randomisations.
- Time only pertains to matching, not I/O or randomisation.
- Test set: ongoing 10th DIMACS challenge on graph partitioning and University of Florida Sparse Matrix Collection.

Results

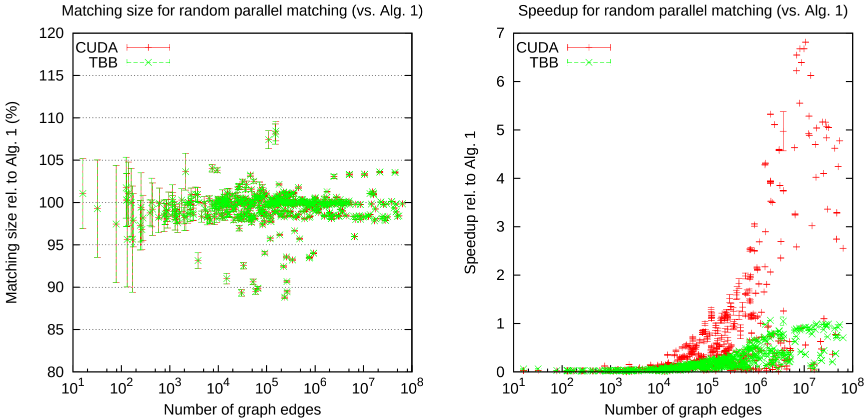
- Created an implementation on the GPU using CUDA and on the CPU using TBB.
- We consider both random and weighted matching.
- Vertex orderings are randomised and results are averaged over 32 randomisations.
- Time only pertains to matching, not I/O or randomisation.
- Test set: ongoing 10th DIMACS challenge on graph partitioning and University of Florida Sparse Matrix Collection.
- Test hardware: dual quad-core Xeon E5620 and an NVIDIA Tesla C2050 (thanks: the Little Green Machine project).

Results (scaling)



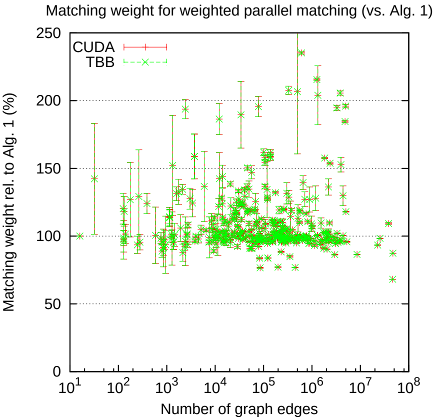
Scaling of TBB implementation (8 physical cores + hyperthreading).

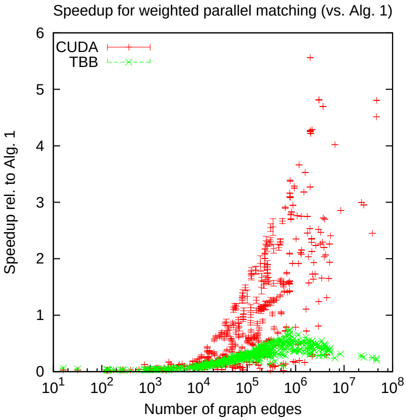
Results (vs. local random matching)



Matching size and speedup for parallel vs. serial local random matching.

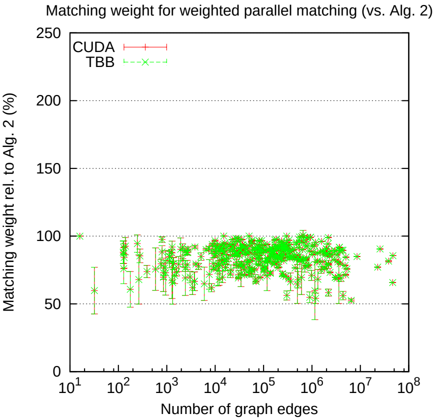
Results (vs. local weighted matching)

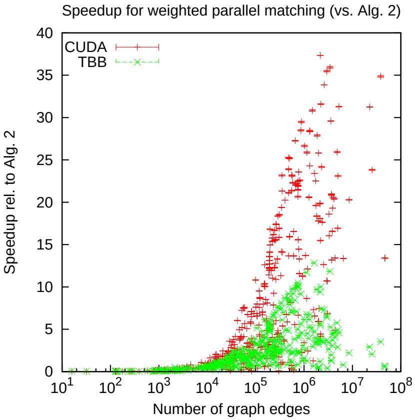




Matching weight and speedup for parallel vs. serial local weighted matching.

Results (vs. global weighted matching)





Matching weight and speedup for parallel local vs. serial global weighted matching.

Conclusion

- We have presented a fine-grain, shared-memory parallel greedy graph algorithm, suited for GPUs.

Conclusion

- We have presented a fine-grain, shared-memory parallel greedy graph algorithm, suited for GPUs.
- The algorithm provides similar quality random matching with speedups up to 6.8 for large graphs.

Conclusion

- We have presented a fine-grain, shared-memory parallel greedy graph algorithm, suited for GPUs.
- The algorithm provides similar quality random matching with speedups up to 6.8 for large graphs.
- The algorithm provides better quality than local weighted matchings with speedups up to 5.6.

Conclusion

- We have presented a fine-grain, shared-memory parallel greedy graph algorithm, suited for GPUs.
- The algorithm provides similar quality random matching with speedups up to 6.8 for large graphs.
- The algorithm provides better quality than local weighted matchings with speedups up to 5.6.
- Compared to a global greedy weighted matching algorithm quality is worse, but speedups up to 37 are achieved.

Conclusion

- We have presented a fine-grain, shared-memory parallel greedy graph algorithm, suited for GPUs.
- The algorithm provides similar quality random matching with speedups up to 6.8 for large graphs.
- The algorithm provides better quality than local weighted matchings with speedups up to 5.6.
- Compared to a global greedy weighted matching algorithm quality is worse, but speedups up to 37 are achieved.
- We look forward to employ this algorithm in (hyper)graph coarsening.

Questions

$\exists$ any questions?

Choosing p

- We should maximise the relative number of matched vertices each round.

Choosing p

- We should maximise the relative number of matched vertices each round.
- The number of matched vertices equals twice the number of **red** vertices that receive at least one proposal: maximise $\frac{2N}{|V|}$, where

$$N := \text{number of } \textbf{red} \text{ vertices receiving at least one proposal.}$$

Choosing p

- We should maximise the relative number of matched vertices each round.
- The number of matched vertices equals twice the number of **red** vertices that receive at least one proposal: maximise $\frac{2\,N}{|V|}$, where

  $N :=$ number of **red** vertices receiving at least one proposal.

- For a random graph with n vertices, we can approximate (independent of edge density)

$$\lim_{n\to\infty} \frac{2\,E(N(n))}{n} \approx 2\,(1-p)\,\left(1 - e^{-\frac{p}{1-p}}\right). \tag{3}$$

Choosing p

Let $G = (\{1, \ldots, n\}, E)$ with $P(\{v, w\} \in E) = d$ for $d \in]0, 1]$. Then $E(N(n))$ is given by

Choosing p

Let $G = (\{1, \ldots, n\}, E)$ with $P(\{v, w\} \in E) = d$ for $d \in]0, 1]$. Then $E(N(n))$ is given by

$$\sum_{v \in V} P(\pi(v) = \textbf{red})\, P(v \text{ is proposed to} \mid \pi(v) = \textbf{red})$$

Choosing p

Let $G = (\{1, \ldots, n\}, E)$ with $P(\{v, w\} \in E) = d$ for $d \in]0, 1]$. Then $E(N(n))$ is given by

$$\sum_{v \in V} P(\pi(v) = \textbf{red})\, P(v \text{ is proposed to} \mid \pi(v) = \textbf{red})$$

$$= \sum_{v \in V} P(\pi(v) = \textbf{red}) \left(1 - \prod_{w \in V \setminus \{v\}} (1 - P(w \text{ proposes to } v \mid \pi(v) = \textbf{red}))\right)$$

Choosing p

Let $G = (\{1, \ldots, n\}, E)$ with $P(\{v, w\} \in E) = d$ for $d \in]0, 1]$. Then $E(N(n))$ is given by

$$\sum_{v \in V} P(\pi(v) = \textbf{red})\, P(v \text{ is proposed to} \mid \pi(v) = \textbf{red})$$

$$= \sum_{v \in V} P(\pi(v) = \textbf{red}) \left(1 - \prod_{w \in V \setminus \{v\}} (1 - P(w \text{ proposes to } v \mid \pi(v) = \textbf{red}))\right)$$

$$= \sum_{v \in V} P(\pi(v) = \textbf{red}) \left(1 - \prod_{w \in V \setminus \{v\}} \left(1 - \frac{P(\pi(w) = \textbf{blue})\, P(\{v, w\} \in E)}{\text{nr. of } \textbf{red} \text{ neighb. of } w}\right)\right)$$

Choosing p

Let $G = (\{1, \dots, n\}, E)$ with $P(\{v, w\} \in E) = d$ for $d \in]0, 1]$. Then $E(N(n))$ is given by

$$\sum_{v \in V} P(\pi(v) = \textbf{red})\, P(v \text{ is proposed to} \mid \pi(v) = \textbf{red})$$

$$= \sum_{v \in V} P(\pi(v) = \textbf{red}) \left(1 - \prod_{w \in V \setminus \{v\}} (1 - P(w \text{ proposes to } v \mid \pi(v) = \textbf{red}))\right)$$

$$= \sum_{v \in V} P(\pi(v) = \textbf{red}) \left(1 - \prod_{w \in V \setminus \{v\}} \left(1 - \frac{P(\pi(w) = \textbf{blue})\, P(\{v, w\} \in E)}{\text{nr. of } \textbf{red} \text{ neighb. of } w}\right)\right)$$

$$\approx n\,(1 - p) \left(1 - \left(1 - \frac{p\,d}{1 + (1 - p)\,(d\,(n - 1) - 1)}\right)^{n-1}\right).$$

NVIDIA visual profiler